Synthesizing efficient systems in probabilistic environments

von Essen, Christian; Jobstmann, Barbara; Parker, David; Varshneya, Rahul

DOI:
10.1007/s00236-015-0237-y

Document Version
Peer reviewed version

Citation for published version (Harvard):
von Essen, C, Jobstmann, B, Parker, D & Varshneya, R 2015, 'Synthesizing efficient systems in probabilistic environments', *Acta Informatica*, vol. 237. https://doi.org/10.1007/s00236-015-0237-y

Acta Informatica manuscript No.
(will be inserted by the editor)

Synthesizing Efficient Systems in Probabilistic Environments

**Christian von Essen, Barbara Jobstmann,
David Parker, Rahul Varshneya**

February 28, 2015

Abstract We present a formalism, algorithms and tools to synthesise reactive systems that behave efficiently, i.e., which achieve an optimal trade-off between a given cost and reward model.

Synthesis aims to automatically generate a program from a specification. Most research in this area focuses on qualitative specifications, i.e., those that define a system as either correct or incorrect. The result can be a system that is correct, but still shows undesired behaviour, e.g., because it is too slow, inefficient or resource-intensive. Quantitative synthesis aims to use additional information to guide the synthesizer towards a desired implementation. Trade-offs between costs and rewards provide a natural source of information in order to guarantee efficiency. The systems we want to synthesize are open, i.e., they react to input signals from their environment. So, we have to specify how to combine the trade-offs the system decides to make for each input. There are several possible ways, e.g., worst or best case, or average case. In this paper we focus on the average case, i.e., we focus on the expected trade-off achieved by a system.

We define the problem of finding the system with the best expected behaviour according to a quantitative specification. This specification associates costs and rewards with each decision the system makes and defines a probabilistic environment that the system operates in. We analyze the feasibility of this task (i.e., prove that such systems exist and are computable) and present three algorithms to compute an optimal system for a given specification. We compare a prototypical implementation of these algorithms against each other and, based on the best-performing algorithm, develop a novel symbolic implementation and integrate it

Christian von Essen
Google Zürich, Zürich, Switzerland

Barbara Jobstmann
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

David Parker
University of Birmingham, Birmingham, UK

Rahul Varshneya
IIT Bombay, India

into the probabilistic model checker PRISM. We report on experiments showing that our algorithm can analyze models with several million states.

1 Introduction

Synthesis aims to automatically generate a program or system from a higher-level specification of its required behaviour. This specification leaves many details unspecified, and it is the synthesizer's task to resolve this non-determinism such that the specification is fulfilled. This higher-level specification allows a programmer or designer to express his wishes concisely, while ignoring implementation details.

This form of abstraction becomes increasingly important as the programs that we write become more complex due to of the arrival of multi-processor systems, heterogeneous systems, increased security requirements and ever more computers in safety-critical systems. The ubiquity of computer software also means that more and more people may benefit from being able to create programs. A high-level language and a synthesizer can lower the bar of creating custom programs.

Program synthesis looks especially promising in the area of embedded systems. Firstly, these systems are often small and less well equipped for interactive development, so debugging becomes especially challenging. Secondly, embedded systems are the most prevalent computer systems today, ranging from thermometers to vehicles on Mars. Finally, embedded systems, by their very nature, have to be customized for each new kind of hardware where they are deployed. Removing unnecessary bugs altogether is therefore desirable and cost-effective.

Embedded systems are also reactive, i.e., they periodically read signals from their environment and write answers to output ports. In this paper, we focus on synthesizing reactive systems [28] from specifications given in temporal logics [33]. In this setting, specifications are usually given with a qualitative meaning, i.e., they classify systems either as good (meaning that the system satisfies the specification) or as bad (meaning that the system violates the specification). Quantitative specifications, on the other hand, assign to each system a value that provides additional information. Traditionally, quantitative techniques are used to analyze properties like response time, throughput or reliability (see, e.g., [13, 20, 3]).

Recently, quantitative reasoning has been used to state preference relations between systems satisfying the same qualitative specification [4]. For example, we can compare systems with respect to robustness, i.e., how reasonably they behave under unexpected behaviours of their environments [6]. A preference relation between systems is particularly useful in synthesis, because it allows the user to guide the synthesizer and ask for "the best" system.

In many settings, though, a better system comes at a higher price. For example, consider an assembly line that can be operated at several speeds, i.e., the number of units produced per time unit. We would prefer a controller that produces as many units as possible. However, running the line in a faster mode increases the power consumption and the probability to fail, resulting in higher repair costs. We are interested in an "efficient" controller, i.e., a system that minimizes the power and repair costs per produced unit. The efficiency of a system is a natural question to ask; it has also been observed by others, e.g., Yue et al. [41] used simulation to analyze energy-efficiency in a MAC (Media Access Control) protocol.

The systems we want to synthesize are open, i.e., they react to signals from their environment. We are looking for a system that is globally optimal, i.e., on all possible behaviours of its environment. Once we have defined a local evaluation criterion, i.e., the efficiency of a system on a single environment input, we need to define a global evaluation criterion. There are several possible ways, e.g., worst-case, best-case or average. In this paper we focus on the average case, i.e., we consider the expected trade-off that a system makes.

To define the average, we assume that the system operates in a probabilistic environment: assembly lines need repairing randomly, network protocols have randomly behaving participants, servers get random requests, etc. Probabilistic modelling allows us to encode knowledge or expectations about the environment's behaviour. Modelling environments with probabilistic behaviour also allows us to assume that the environment is not hostile, i.e., it is not in fact trying to do its worst. Embedded systems are sometimes only required to operate in certain conditions, which we can model probabilistically. A server, for example, is only required to work given an expected average number of requests, whereas a DDOS attack lies outside of its specification[1]. A different component takes care of shielding the server in case of such an attack. Lastly, assuming probabilistic behaviour can make quantitative synthesis questions more tractable than their qualitative counterparts, admitting synthesis algorithms with expected polynomial instead of exponential run-time.

Finally, we note that the system and its environment co-exist in a closed loop, both able to impact the behaviour of the other. In the case of assembly lines, for example, failing lines certainly influence the behaviour of their controller. Conversely, the reactions of the controller influence the probability of lines failing.

In this paper we show how to automatically synthesize a system that has an "efficient" average-case behaviour in a given environment. The Oxford English Dictionary defines the adjective efficient as: *"(of a system or machine) achieving maximum productivity with minimum wasted effort"*. We analogously define efficiency as the ratio between a given *cost* model and a given *reward* model. To further motivate this choice, consider the following example: assume we want to implement an automatic gear-shifting unit (Automatic Clutch and Throttle System - ACTS) that optimizes its behaviour for a given driver profile. The goal of our implementation is to optimize the fuel consumption per kilometer (l/km), a commonly used unit to advertise efficiency. In order to be most efficient, our system has to maximize the speed (given in km/h) while minimizing the fuel consumption (measured in liters per hour, i.e., l/h) for the given driver profile. If we take the ratio between the fuel consumption (the "cost") and the speed (the "reward"), we obtain l/km, the desired measure.

Given an efficiency measure, we ask for a system with an optimal average-case behaviour. The average-case behaviour with respect to a quantitative specification is the expected value of the specification over all possible behaviours of the system in a given probabilistic environment [11]. We describe the probabilistic environment using Markov decision processes (MDPs), which is a more general model than the one considered in [11]. It allows us to describe environments that react to the behaviour of the system (like the driver profile).

[1] We can model a DDOS attack, e.g., by assuming that it happens with low probability: on attack, the environment suddenly changes its behaviour drastically.

Our contributions can be summarized as follows:

1. We present a framework to automatically construct a system with an efficient average-case behaviour with respect to a reward and a cost model in a probabilistic environment. To the best of our knowledge, this is the first approach that allows automatic synthesis of such systems. We introduce our framework using a simple example in Section 2, explaining necessary definitions along the way. We analyze our framework in Section 3, proving that we can indeed find optimal systems. This analysis is the foundation for the algorithms to come.
2. We present three algorithms to compute systems of optimal efficiency. These are described in Section 4. All three start by decomposing the MDP into end components [13], but differ in the way they compute an optimal strategy for each one. The first algorithm uses fractional linear programming. The second, a simple adaption of an algorithm presented in [13], is based on a reduction to linear programming. The third algorithm is based on policy iteration and a sequence of reductions to MDPs with long-run average-reward objective.
3. We present a semi-symbolic variant of the policy iteration algorithm, which combines symbolic (binary decision diagram-based) and explicit-state techniques for efficiency. Details are in Section 5.
4. We have implemented all algorithms in a standalone tool and compare them on our examples in their respective sections. In order to increase the scope of our approach, we also integrated the best-performing explicit algorithm and the symbolic algorithm into PRISM [24], a well-known probabilistic model checker. We report experiments with this implementation, and show that our algorithm can analyze models with several million states.

Parts of this paper (contributions 1 and 2) are based on [37] and [38].

1.1 Related Work

Related work can be divided into two categories: (1) work using MDPs for quantitative synthesis and (2) work on MDP reward structures.

For the former, we first consider the work of Chatterjee et al. [11]. We generalize this work in two directions: (i) we consider ratio objectives, a generalization of average-reward objectives and (ii) we introduce a more general environment model based on MDPs that allows the environment to change its behaviour based on actions the system has taken. In the same category, there is the work of Parr and Russell [32], who use MDPs with weights to present partially specified machines in Reinforcement Learning. Our approach differs from this approach, as we allow the user to provide the environment, the specification, and the objective function separately and consider the expected ratio reward, instead of the expected discounted total reward, which allows us to ask for efficient systems. We also mention multi-objective verification techniques for MDPs, which consider multiple quantitive specifications using, e.g., linear temporal logic [12,15] or long-run rewards [7] but these are not able to reason about ratios, as we do here. Finally, in [39], Wimmer et al. introduce a semi-symbolic policy algorithm for MDPs with the average objective, while we present a semi-symbolic policy algorithm for MDPs with the ratio objective, subsuming the former.

Semi-MDPs [34] fall into the second category. Unlike work based on Semi-MDPs, we allow a reward of value 0. Furthermore, we provide an efficient policy iteration algorithm that works on our Ratio-MDPs as well as on Semi-MDPs. Approaches using the discounted reward payoff (cf. [34]) are also related but focus on immediate rewards instead of long-run rewards. Similarly related is the work of Derman [14], who considered the payoff function obtained by dividing the expected costs by expected rewards. These two objective functions are in general not the same. We believe that our payoff function is more natural as we will show in Section 3.4. The closest to our work is that of de Alfaro [13]. In this work, the author also allows rewards with value 0, and he defines the expected payoff over all runs that visit a reward with value greater than zero infinitely often. In our framework the payoff is defined for all runs. De Alfaro also provides a linear programming solution, which can be used to find the ratio value in an end component (see Section 7). We provide two alternative solutions for end components including an efficient policy iteration algorithm. Finally, we are the first to implement and compare these algorithms and use them to synthesize efficient controllers.

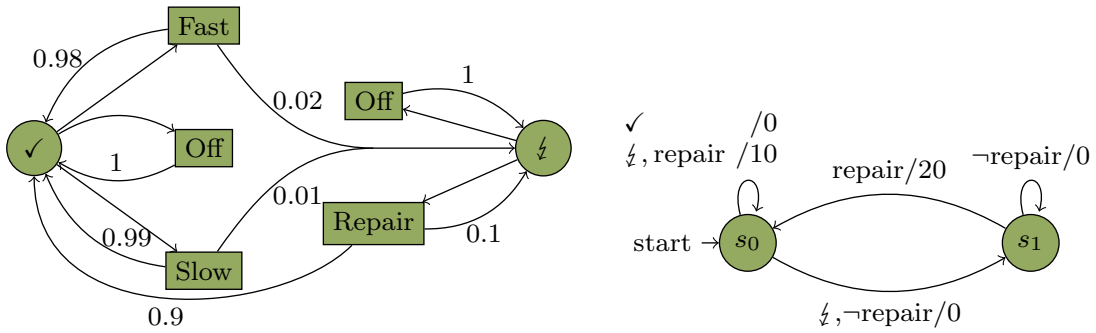
2 The System and its Environment

In this section we will introduce the system, its environment and the quantitative monitor. We will show how they operate in lockstep and how their combination leads to a system with measurable performance. While doing so, we will introduce the necessary notation and definitions.

2.1 The System

The systems we aim to synthesize are *reactive systems*. That is, systems that react infinitely to events from their environment. As usual, we model a reactive system as a *Moore machine*, i.e., a machine that reads *letters* from an *alphabet* as *input* and writes letters in turn as *output*. Given a finite set of *symbols* or *atomic propositions* AP, we form a set of letters $\Sigma = 2^{AP}$ as an alphabet. We denote by Σ^* the set of finite words over that alphabet, i.e., the set of finite sequences of those letters. Analogously, by Σ^ω we denote the set of infinite words over Σ, i.e., the set of infinite sequences of those letters. Given a word $w \in \Sigma^* \cup \Sigma^\omega$, we denote by $|w|$ the length of the word and by w_i the i-th letter of that word for $0 \leq i < |w|$, i.e., we start counting from zero.

Definition 1 (Transducer) A transducer is a tuple $T = (S, s_0, I, O, \delta, \gamma)$. By S we denote the finite set of states of T, by s_0 its initial state, by $I \subseteq AP$ its input alphabet and by $O \subseteq AP$ its output alphabet. Sets I and O form a partition of AP, i.e., $I \cup O = AP$ and $I \cap O = \emptyset$. Function $\delta : S \times 2^I \to S$ is the *transition function* of T, defining how it moves from state to state in the course of reading its input. Finally, function $\gamma : S \times 2^I \to 2^O$ is the *output function* of T, defining what output it writes, given the current state and the current input letter. If γ is constant in its second parameter (i.e., if $\forall s \in S\ \forall i_0, i_1 \in 2^I : \gamma(s, i_0) = \gamma(s, i_1)$), then we call T a *Moore machine*, otherwise it is a *Mealy machine*. For Moore machines, we sometimes use $\gamma : S \to 2^O$ and $\gamma : S \times 2^I \to 2^O$ equivalently.



(a) An Environment (b) A monitor

Fig. 1 Environment model and quantitative specification of the production line example

Given an infinite input word $w^I \in (2^I)^\omega$, the definition of T implies a *run* $\rho = s_0 s_1 \ldots$ of T on w defined by $s_i = \delta(s_{i-1}, w^I_{i-1})$ for $i > 0$. Analogously, it implies an output word $w^O \in (2^O)^\omega$ by $w^O_i = \gamma(s_i, w^I_i)$. The *combined input-output word* w is defined by $w_i = w^I_i \cup w^O_i$ for all $i \geq 0$.

Consider a production plant that has several lines that can be turned on and off. A system (i.e., the plant controller) here reads the state of the production lines (e.g., if each one is broken or working). It then decides to turn specific lines on or off based on this state information.

We model the events from the environment as an infinite stream of input letters, and the reactions of the system as an infinite stream of output letters. It is our goal to enable the probabilistic environment to react to the reactions of the system. To that end, we make the output of the system the input of the environment, thus forming a feedback loop. The system and the environment are stateful. Depending on the reactions of the system, the environment changes its state. We model such an environment as a *Markov decision process*.

Definition 2 (Markov decision process) A *Markov decision process (MDP)* is a tuple $\mathcal{M} = (M, m_0, A, \overline{A}, p)$, where M is the finite set of states of $\mathcal{M}$, $m_0 \in M$ is the initial state, $A \subseteq 2^O$ is the set of *actions*, $\overline{A} \subseteq M \times A$ is the action activation relation and $p : M \times A \times M \to [0, 1]$ is the probability transition function, i.e., we require that $\sum_{m' \in M} p(m, i, m') = 1$ for all states $m \in M$ and actions $i \in A$. We also demand that each state has at least one activated action, i.e., that $\forall m \in M\ \exists a \in A :\ (m, a) \in \overline{A}$. When using an MDP to model the environment, we assume without loss of generality that all actions are always activated, i.e., $\overline{A} = M \times A$. For convenience, we also write $\overline{A}(m)$ to denote the set $\{a \in A \mid (m, a) \in \overline{A}\}$.

A *Markov chain (MC)* is a Markov decision process for which there exists exactly one action for each state, i.e., for which the cardinality of the set $\overline{A}(m)$ is one for all states $m \in M$. For a Markov chain $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ we sometimes write $\mathcal{M} = (M, m_0, p)$, and then we also write $p : M \times M \to [0, 1]$.

A *labelling* for $\mathcal{M}$ is a function $\lambda : M \to 2^I$ that is deterministic with respect to the transition function of $\mathcal{M}$, i.e., for all states $m, m', m'' \in M$ and every action $a \in A$ such that $p(m, a, m') > 0$ and $p(m, a, m'') > 0$ and $m' \neq m''$ we have $\lambda(m') \neq \lambda(m'')$. The labelling intuitively allows us to decouple model states from inputs the model feeds the system. We could do without it, but it occasionally makes describing a model more pleasant.

2.2 The Environment

Example 1 (Modelling a single production line) The model of a single production line is shown in Figure 1(a). A line has two states: broken (ϟ) and ok (✓). In each of these states, the system can either turn a production line on to a slow mode with action Slow, turn it on to a fast mode with action Fast, switch it off with action Off, or repair it with action Repair. The failure of a production line is controlled by the environment. We assume a failure probability of 1% when the production line is running slowly and 2% when the production line is running fast. If it is turned off, then a failure is impossible. Transitions in Figure 1(a) are labelled with actions and probabilities, e.g., the transition from state ✓ to ✓ labelled with action Slow and probability 0.99 means that we go from state ✓ with action Slow with probability 0.99 to state ✓. Note that the labels of the states (✓ and ϟ) of this MDP correspond to decisions the environment can make. The actions of the MDP are the decisions the system can use to control the environment. The specification for n production lines is the synchronous product of n copies of the model in Figure 1(a), i.e., the state space of the resulting MDP is the Cartesian product, and the transition probabilities are the product of the probabilities; for example, for two production lines, the probability to move from (✓ , ✓) to (✓ , ✓) on action (Slow , Slow) is 0.99^2.

The system and its environment now form a feedback loop, as depicted in Figure 2: First, $\mathcal{M}$ (the Environment) signals its current (initial) state to T (the System). Then, T changes its state and provides an output letter, based on its own state and the state of $\mathcal{M}$. The environment $\mathcal{M}$ will read that letter, change its state probabilistically, and then provide the next output letter. The system reads this letter, changes its state, and provides the next letter. $\mathcal{M}$ reads this letter, makes a probabilistic choice based on it and its current state, and provides the next letter, and so on ad infinitum. This loop allows us to model control over the environment by the system.

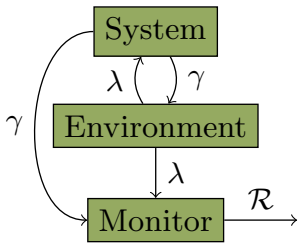


Fig. 2 Overview

2.3 The Monitor

The task of the monitor will be to measure the stream of states of the environment and the outputs of the system. We model the monitor as a transducer, but one whose output we fix to be two real numbers. These numbers model the *cost* and *reward* of the decisions of the system.

Definition 3 (Monitor) A monitor $\mathcal{O} = (O, o_0, I_\mathcal{O}, O_\mathcal{O}, \delta_\mathcal{O}, \gamma_\mathcal{O})$ for an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ is a transducer that reads letters from $I_\mathcal{O} = M \times A$ as input and writes pairs of positive real values (i.e., $O_\mathcal{O} = \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$) as output. We sometimes write $c, r : O \times I_\mathcal{O} \to \mathbb{R}^{\geq 0}$ for the first and second components of $\gamma_\mathcal{O}$.

We use a monitor to evaluate a system with respect to a desired property. It reads words over the joint input/output alphabet and assigns a value to them. For example, the monitor for the production line controlling system reads pairs consisting of (i) a state of a production line (input of the system) and (ii) an action

(output of the system). We obtain this transducer by composing transducers with a single cost function in various ways.

Example 2 (Monitor of a production line) In our example, we use for each production line two transducers with a single cost function to express the repair costs and the production due to this line. The transducer for the repair costs is shown in Figure 1(b). It assigns repair costs of 10 for repairing a broken production line immediately and costs 20 for a delayed repair. If we add the numbers the transducer outputs, we obtain the repair costs of a run. For example, sequence (✓, Slow) (↯, Repair) (↯, Repair) has cost $0+10+10=20$. The amount of units depends on the speed of the production line. The transducer describing the number of units produced assigns value 2 if a production line is running on slow speed, 4 if it is running on fast speed, and 0 if the production line is turned off or broken.

We extend the specification to multiple production lines by building the synchronous product of copies of the transducer described above and compose the cost and reward functions in the following ways: we sum the rewards for the production and we take the maximum of repair costs of different production lines to express a discount for simultaneous repairs of more than one production line. The final specification transducer is the product of the production automaton and the repair cost automaton with (i) the repair cost as cost function and (ii) the measure of productivity as reward function.

In the current specification a system that keeps all production lines turned off has the (smallest possible) value zero, because lines that are turned off do not break down and repair is unnecessary. Therefore, we require that at least one of the lines is working. We can specify this requirement by using a qualitative specification described by a safety automaton[2]. This safety requirement can then be ensured by adapting the cost functions of the ratio objective [11,37]. For simplicity, we say here that any action in which all lines are turned off has an additional cost of 10.

2.4 Combining System, Environment and Monitor

In Section 2.2 we described how System and Environment work together. Now, in addition, the system provides its output and the environment its state to the monitor. The monitor then provides two numbers in a tuple. These numbers model the cost and reward of the decision the system made in the current context. We describe this collaboration graphically in Figure 2. We now combine these three into one object as follows.

Definition 4 (Combination of System, Environment and Monitor) We define the *Extended MDP* of Environment $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ with a Monitor $\mathcal{O} = (O, o_0, I_{\mathcal{O}}, O_{\mathcal{O}}, \delta_{\mathcal{O}}, \gamma_{\mathcal{O}})$ to be the MDP $\mathcal{M}' = (M', m_0', A', \overline{A}', p')$, where $M' = M \times O$ is its set of states, $m_0' = (m_0, o_0)$ is its start state, $A' = A$ is its set of actions, $\overline{A}' = \overline{A}$ is its action activation function, and $p' : M' \times A' \times M' \to [0,1]$ is its probabilistic transition function, where $p'((m,o),a,(m',o')) = p(m,a,m')$ if $o' = \delta_{\mathcal{O}}(o,(m,a))$ and zero otherwise.

[2] Our approach can also handle liveness specifications resulting in a Ratio-MDP with parity objective, which is then reduced to solving a sequence of MDPs with mean-payoff parity objectives [11].

This combination also induces an output function $\gamma_{\mathcal{M}} : M' \times A \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ of the MDP. The output function is defined as the output of the monitor in the same context, i.e., $\gamma_{\mathcal{M}}((m, o), a) = \gamma_{\mathcal{O}}(o, (m, a))$. As for monitors, we often use $c, r : C \to \mathbb{R}^{\geq 0}$ as shorthand for the cost and reward parts of $\gamma_{\mathcal{M}}$.

The *Combination* of System $T = (S, s_0, I, O, \delta, \gamma)$, Environment $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ with labelling λ and Monitor $\mathcal{O} = (O, o_0, I_{\mathcal{O}}, O_{\mathcal{O}}, \delta_{\mathcal{O}}, \gamma_{\mathcal{O}})$ is defined as a *Markov chain*, i.e., as a tuple $\mathcal{C} = (C, c_0, p_{\mathcal{C}})$, where $C = S \times M \times O$ is its set of states, $c_0 = (s_0, m_0, o_0)$ is its initial state and $p_{\mathcal{C}} : C \times C \to [0, 1]$ is its probabilistic transition function.

The probabilistic transition function $p_{\mathcal{C}}$ models the progression of System, Environment and Monitor in lockstep, i.e., $p_{\mathcal{C}}((s, m, o), (s', m', o')) = p(m, \gamma(s), m')$ if $s' = \delta(s, \lambda(m))$ is the next state of T, based on its current state and the labelling of the state of the environment, and $o' = \delta_{\mathcal{O}}(o, (m, \gamma(s)))$ is the next state of the monitor, based on its current state, the state of the Environment and the output of the system. Otherwise the value of $p_{\mathcal{C}}$ is zero.

This combination also induces an output function $\gamma_{\mathcal{C}} : C \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ on the Markov chain. The output function is defined as the output of the monitor in the same context, i.e., $\gamma_{\mathcal{C}}(s, m, o) = \gamma_{\mathcal{O}}(o, (m, \gamma(s)))$. As for monitors, we often use $c, r : C \to \mathbb{R}^{\geq 0}$ as shorthand for the first and second part of $\gamma_{\mathcal{C}}$.

We sometimes interpret the probabilistic transition function as a matrix, i.e., we enumerate the state space from 1 to $n := |C|$, and interpret $p_{\mathcal{C}}$ as an $n \times n$ matrix, where the entry in row i and column j has value $p_{\mathcal{C}}(c_i, c_j)$. Analogously, we can interpret every function $f : C \to \mathbb{R}$ as a row or column vector of dimension n, where entry i has value $f(c_i)$.

Example 3 (State transition probabilities of production lines) This combination provides us with a probability distribution over the development of System, Environment and Monitor over time. Consider the case of two production lines. The probability of moving, e.g., from $((\checkmark, \checkmark), (s_0, s_0))$ to $((\lightning, \lightning), (s_0, s_0))$ when choosing (Slow, Slow) is 0.01^2, while the probability of moving to $((\lightning, \lightning), (s_1, s_1))$ is 0 because we cannot move from s_0 to s_1 with this input.

2.5 Measuring Efficiency

While we now have a way to measure local decisions, we are still lacking a means to measure the global, long-run quality[3] of the system. To that end, we will first define how to evaluate the efficiency of a single run of the composition, and we will then hint at how to evaluate the efficiency of the whole composition.

Definition 5 (Efficiency/Ratio of a Run) Let $\rho \in C^\omega$ be an infinite run of a Combination and let $c, r : C \to \mathbb{R}^{\geq 0}$ be two functions. We then define the *efficiency* or *ratio* of the run as

$$\mathcal{R}_{\frac{c}{r}}(\rho) = \lim_{l \to \infty} \liminf_{u \to \infty} \frac{\sum_{i=l}^{u} c(\rho_i)}{1 + \sum_{i=l}^{u} r(\rho_i)}$$

We call $\mathcal{R}_{\frac{c}{r}}$ the *Ratio Payoff Function*. We often leave out c and r if they are clear from the context, just writing $\mathcal{R}$. Intuitively, $\mathcal{R}$ computes the long-run ratio between the costs and rewards accumulated along a run. We divide costs by rewards,

[3] Pun intended

i.e., the higher the efficiency of a run, the lower the ratio. Therefore, in the rest of this paper we try to minimize the ratio.

The first limit allows us to ignore a finite prefix of the run, which ensures that we only consider the long-run behaviour. The 1 in the denominator avoids division by 0 if the accumulated costs are 0 and has no effect if the accumulated costs are infinite. We need the limit inferior here because the sequence of the limit might not converge.

Consider a Combination with states s and t, and the run $\rho = s^1t^2s^4t^8s^{16}\dots$, where s^k means that state s is visited k times. Assume state s and state t have the following costs: $c(s) = 0$, $r(s) = 1$, $c(t) = 1$ and $r(t) = 1$. Then, the efficiency of $\rho_0 \dots \rho_i$ will alternate between $1/6$ and $1/3$ with increasing i and hence the sequence for $i \to \infty$ will not converge. The limit inferior of this sequence is $1/6$.

There is a special case of this ratio function in which the co-domain of the reward function r is $\{1\}$. This leads us to the classic *Mean Payoff Function* [34].

Definition 6 (Mean Cost of a Run [34]) Let $\rho \in C^\omega$ be a run of a Combination and let $c : C \to \mathbb{R}^{\geq 0}$ be a cost function. We then define the *mean cost* $\mathcal{P}_c(\rho)$ of ρ with the help of a reward function $r_1(c) = 1$ for all states c. Formally, $\mathcal{P}_c(\rho) = \mathcal{R}_{\frac{c}{r_1}}(\rho)$.

Since we are aiming at optimal average efficiency, we need the expected value of $\mathcal{R}$ over all runs. To do so, we interpret the runs of the Combination $\mathcal{C}$ as an MC, and then use the standard definition of a probability measure $\mu_{\mathcal{C}}$ over these runs [22]. Given a measurable function f over runs, we use $\mathbb{E}_{\mathcal{C}}[f]$ to denote the expected value of f under $\mu_{\mathcal{C}}$. The efficiency of a Combination of System, Environment and Monitor is then the expected efficiency of all its runs.

Definition 7 (Efficiency/Ratio of a Combination) Given a Combination $\mathcal{C}$, we define the efficiency or ratio of the combination as $\mathbb{E}_{\mathcal{C}}[\mathcal{R}]$.

Lemma 1 (Expected Ratio Exists) *The expected ratio $\mathbb{E}_{\mathcal{C}}[\mathcal{R}]$ exists since $\mathcal{R}$ is bounded from below by zero.*

We now can ask for an efficient system in a probabilistic environment. We model the system and the monitor as transducers and the environment as an MDP. We evaluate the performance of a system in this context as its expected efficiency, modelled by the expected ratio of costs and rewards. In the next section we will analyze the combination of the three components and show the theory necessary to find the optimal system for an environment and a monitor.

3 Analysis

In this section, we will lay the foundations of the algorithms in Section 4. We will introduce strategies, that is, functions resolving the non-determinism of an MDP, based on the sequence of states the MDP has visited so far. We will show that systems and strategies are one and the same. After having established that, we will show that a class of simple strategies is sufficient, i.e., that we can always find an optimal strategy in this class. Thus we will make our search for the most efficient system simpler. We will further show how to calculate the expected ratio of the simple strategies. On the basis of these results, we will look for algorithmic solutions to this search in Section 4.

3.1 Strategies and Systems

To find a system T such that the combination of T, Environment $\mathcal{M}$ and monitor $\mathcal{O}$ is optimal, we combine $\mathcal{M}$ and $\mathcal{O}$ to obtain a new MDP as described in Definition 4. We will then look for an optimal *strategy* in the resulting MDP.

Definition 8 (Strategy) A *strategy* (or *policy*) for an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ is a function $d : (M \times A)^* M \to \mathcal{D}(A)$, where $\mathcal{D}(A) = \{f : A \to [0,1] \mid \sum_{a \in A} f(a) = 1\}$ is the set of all *probability distributions* over A. This function assigns a probability distribution to all finite sequences in $(M \times A)^* M$ such that only active actions are chosen, i.e., for all sequences $w \in (M \times A)^*$, states $m \in M$ and actions $a \in A$ such that $d(wm)(a) > 0$ we have $(m, a) \in \overline{A}$.

A strategy such that the co-domain of $d(\rho)$ is $\{0, 1\}$ for all $\rho \in (M \times A)^* M$ is called *deterministic*, in which case we write it as a function of the form $d : (M \times A)^* M \to A$. A strategy that can be defined using domain M is called *memoryless*. A memoryless, deterministic strategy is called *pure* and takes the form $d : M \to A$. We denote the set of pure strategies by $D(\mathcal{M})$.

We can use transducers to describe an important class of strategies.

Definition 9 (Finite-memory strategies) Let $d : (M \times A)^* M \to A$ be a strategy for an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$. If there is a transducer $T = (S, s_0, I, O, \delta, \gamma)$ such that $d(m_0) = \gamma(m_0)$ and $d(m_0 a_0 m_1 a_1 \dots m_n) = \gamma(s_n)$ for all $n \geq 0$, where $s_n = \delta(s_{n-1}, m_{n-1})$, then we say d is a *finite-memory* or *finite-state* strategy.

Like transducers in Definition 4, pure strategies induce Markov chains.

Definition 10 (Induced Markov Chain) Let $\mathcal{M}$ be an MDP and d be a pure strategy for $\mathcal{M}$. Then by $\mathcal{M}_d = (C, c_0, p_{\mathcal{C}})$ we denote the *induced Markov chain*, where $\mathcal{M}$ and $\mathcal{M}_d$ have the same set of states, i.e., $C = M$, the same start, i.e., $c_0 = m_0$, and the probability function is defined by d, i.e., $p_{\mathcal{C}}(c, c') = p(c, d(c), c')$ for all states $c, c' \in C$.

The following lemma states that for every pure strategy (i.e., a function getting states as input) there is a transducer (i.e., a function getting sequences of labels as input), such that the two induce the same Markov chain and therefore the same expected ratio. The lemma follows from the determinism of the labeling.

Lemma 2 (Relation between Transducers and Pure Strategies) *Let $\mathcal{M}$ be an MDP and let $d : M \to A$ be a pure strategy for $\mathcal{M}$. Then for any ratio function $\mathcal{R}$ there is a transducer T such that for the combination $\mathcal{C}$ of T and $\mathcal{M}$ we have that $\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \mathbb{E}_{\mathcal{M}_d}[\mathcal{R}]$.*

We are looking for an optimal pure strategy for the MDP constructed from the Environment model and the Monitor. In the next subsection we will show that there always exists an optimal pure strategy.

3.2 Pure Strategies are Sufficient

In [18], Gimbert proves that in an MDP any function mapping sequences of states of that MDP to $\mathbb{R}$ that is *submixing* and *prefix independent* admits optimal pure

strategies. Since $\mathcal{R}$ maps only to non-negative values and the set of measurable functions is closed under addition, multiplication, limit inferior and superior and division, provided that the divisor is not equal to 0, the expected value of $\mathcal{R}$ is always defined and the theory presented in [18] also applies in our case. To adapt the proof of [18] to minimizing the function instead of maximizing it, one only needs to inverse the used inequalities and replace max by min. It remains to show that $\mathcal{R}$ fulfills the following two properties.

Lemma 3 ($\mathcal{R}$ is submixing and prefix independent) *Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be a MDP and ρ be a run.*

1. *For every $i \geq 0$ the prefix of ρ up to i does not matter, i.e., $\mathcal{R}(\rho) = \mathcal{R}(\rho_i \rho_{i+1} \dots)$.*
2. *For every sequence of non-empty runs $u_0, v_0, u_1, v_1 \dots \in (A \times M)^+$ such that $\rho = u_0 v_0 u_1 v_1 \dots$ we have that the function of the sequence is greater than or equal to the minimal ratio of sequences $u_0 u_1 \dots$ and $v_0 v_1 \dots$, i.e., $\mathcal{R}(\rho) \geq \min\{\mathcal{R}(u_0 u_1 \dots), \mathcal{R}(v_0 v_1 \dots)\}$.*

Proof The first property follows immediately from the first limit in the definition of $\mathcal{R}$. For the second property we partition $\mathbb{N}$ into U and V such that U contains the indexes of the parts of ρ that belong to a u_k for some $k \in \mathbb{N}$ and such that V contains the other indexes. Formally, we define $U := \bigcup_{i \in \mathbb{N}} U_i$ where $U_0 := \{k \in \mathbb{N} \mid 0 \leq k < |u_0|\}$ and $U_i := \{\max(U_{i-1}) + |v_{i-1}| + k \mid 1 \leq k \leq |u_i|\}$. Let $V := \mathbb{N} \setminus U$ be the other indexes.

Now we look at the value from m to l for some $m \leq l \in \mathbb{N}$, i.e. $\mathcal{R}^l_m := (\sum_{i=m \dots l} c(\rho_i))/(1 + \sum_{i=m \dots l} r(\rho_i))$. We can divide the sums into two parts, the one belonging to U and the one belonging to V and we get

$$\mathcal{R}^l_m = \frac{\left(\sum_{i \in \{m \dots l\} \cap U} c(\rho_i)\right) + \left(\sum_{i \in \{m \dots l\} \cap V} c(\rho_i)\right)}{1 + \left(\sum_{i \in \{m \dots l\} \cap U} r(\rho_i)\right) + \left(\sum_{i \in \{m \dots l\} \cap V} r(\rho_i)\right)}$$

We now define the sub-sums between the parentheses as $u_1 := \sum_{i \in \{m \dots l\} \cap U} c(\rho_i)$, $u_2 := \sum_{i \in \{m \dots l\} \cap U} r(\rho_i)$, $v_1 := \sum_{i \in \{m \dots l\} \cap V} c(\rho_i)$ and $v_2 := \sum_{i \in \{m \dots l\} \cap V} r(\rho_i)$. Then we obtain

$$\mathcal{R}^l_m = \frac{u_1 + v_1}{1 + u_2 + v_2}$$

We will now show

$$\mathcal{R}^l_m \geq \min\left\{\frac{u_1}{u_2 + 1}, \frac{v_1}{v_2 + 1}\right\}$$

Without loss of generality we can assume $u_1/(u_2 + 1) \geq v_1/(v_2 + 1)$, then we have to show that

$$\frac{u_1 + v_1}{1 + u_2 + v_2} \geq \frac{v_1}{v_2 + 1}.$$

This holds if and only if $(u_1 + v_1)(1 + v_2) = u_1 + v_1 + u_1 v_2 + v_1 v_2 \geq v_1 + v_1 u_2 + v_1 v_2$ holds. By subtracting v_1 and $v_1 v_2$ from both sides we obtain $u_1 + u_1 v_2 = u_1(1 + v_2) \geq u_2 v_1$. If u_2 is equal to 0 then this holds because u_1 and v_2 are greater than or equal to 0. Otherwise, this holds if and only if $u_1/u_2 \geq v_1/(1+v_2)$ holds. In general, we have $u_1/u_2 \geq u_1/(u_2+1)$. From the assumption we have $u_1/(u_2+1) \geq v_1/(v_2+1)$

and hence $u_1/u_2 \geq v_1/(v_2+1)$. The original claim follows because we have shown this for any pair of m and l. □

Theorem 1 (There is always a pure optimal strategy) *For each MDP with the ratio function, there is a pure optimal strategy.*

Proof See [18] and the last lemma. □

This theorem allows us to restrict the search for an optimal strategy (and therefore optimal system) to a finite set of possibilities. In the next subsection, we show how to calculate the expected ratio of a pure strategy. Then, in the next section, we will show algorithms that perform better than brute force search.

3.3 Expected Ratio of Pure Strategy

To calculate the expected value of a pure strategy, we use the fact that an MDP with a pure strategy induces a Markov chain and that the runs of a Markov chain have a special property, which we can use to calculate the expected value. We will first show how to calculate the expected value on a *unichain* MC, and will then extend the result to any kind of Markov chain.

Definition 11 (Random variables of MCs [34]) Let $p^n_{\mathcal{C}}(c)$ be the probability of being in state c at step n and let $\pi(c) := lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} p^i_{\mathcal{C}}(c)$. This is called the *steady state distribution* of $p^n_{\mathcal{C}}$. Let ν^n_c denote the *number of visits* to state c up to time n.

Definition 12 (Recurrent/Transient States, Recurrence Class, Unichain [34]) Let $\mathcal{C}$ be a Markov chain. A state $c \in C$ is called *transient* if the probability of it occurring infinitely often in a run of $\mathcal{C}$ is equal to zero; otherwise it is called *recurrent*. A subset of states S of $\mathcal{C}$ is called a *Recurrence Class* if all states can reach each other, all states are recurrent, and there is no such set of states S' such that $S \subset S'$. We say that a Markov chain is *Unichain* if it has at most one recurrence class. We call an MDP unichain if every strategy induces a unichain MC.

We have the following lemma describing the long-run behaviour of Markov chains [36,30].

Definition 13 (Well-Behaved runs) Let ρ be an infinite run of a Markov chain. Then we call this run *well-behaved* if $\lim_{l\to\infty} \frac{\nu^l_c}{l} = \pi(c)$.

Lemma 4 (Runs are Well-Behaved Almost Surely [34]) *Runs of a Markov chain are well-behaved almost surely, i.e.,* $\mathrm{P}(\textit{well-behaved}) = 1$.

When we calculate the expected ratio, we only need to consider well-behaved runs as shown in the following lemma.

Lemma 5 *Let $\mathcal{C} = (C, c_0, p_{\mathcal{C}})$ be a Markov chain, let $\mathcal{P} = (\Omega, \mathcal{F}, \mu)$ denote its induced probability space, and let N denote the set of runs that are not well-behaved. Then*

$$\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \int_{\Omega \setminus N} \mathcal{R} \; d\mu$$

Proof The expected value is defined as $\mathbb{E}[\mathcal{R}] = \int_\Omega \mathcal{R}\, d\mu$. According to a well known property of Lebesgue integrals, we can ignore events having probability 0 when calculating the integral, i.e., $\int_\Omega \mathcal{R}\, d\mu = \int_{\Omega \setminus N} \mathcal{R}\, d\mu$ for any set of events N with $\mu(N) = 0$. From Lemma 4, it follows that the set of runs that are not well-behaved has probability zero. □

For a well-behaved run, i.e., for every run that we need to consider when calculating the expected value, we can calculate the ratio in the following way.

Lemma 6 (Calculating the Ratio of a Well-Behaved Run) *Let ρ be a well-behaved run of a unichain Markov chain $\mathcal{C} = (C, c_0, p_\mathcal{C})$ and let $\gamma_\mathcal{C} : C \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ be an output function. Recall that we denote by c the first cost of $\gamma_\mathcal{C}$, and by r the reward.*

$$\mathcal{R}(\rho) = \frac{\sum_{c \in C} \pi(c) c(c)}{\lim_{l \to \infty} \frac{1}{l} + \sum_{c \in C} \pi(c) r(c)}$$

Proof By definition of $\mathcal{R}$ we have

$$\mathcal{R}(\rho) = \lim_{l \to \infty} \liminf_{m \to \infty} \frac{\sum_{i=l}^{m} c(\rho_i)}{1 + \sum_{i=l}^{m} r(\rho_i)}$$

To get rid off the outer limit, we are going to assume, without loss of generality, that there are no transient states. We can do this because every transient state will not influence $\mathcal{R}(\rho)$ because ρ is well-behaved and because $\mathcal{R}$ is prefix independent.

$$\mathcal{R}(\rho) = \liminf_{l \to \infty} \frac{\sum_{i=0}^{l} c(\rho_i)}{1 + \sum_{i=0}^{l} r(\rho_i)}$$

We can calculate the sums in a different way: we take the sum over the states and count how often we visit one state, i.e.,

$$\frac{\sum_{i=0}^{l} c(\rho_i)}{1 + \sum_{i=0}^{l} r(\rho_i)} = \frac{\sum_{c \in C} c(c) \nu_c^l}{1 + \sum_{c \in C} r(c) \nu_c^l} = \frac{\sum_{c \in C} c(c)(\nu_c^l / l)}{1/l + \sum_{c \in C} r(c)(\nu_c^l / l)}$$

We will now show that the sequence converges for lim instead of lim inf. But if a sequence converges for lim, then it also converges to lim inf, and the two limits have the same value. Because both the numerator and the denominator are finite values we can safely draw the limit into the fraction, i.e.,

$$\begin{aligned}
(\dagger) \lim_{l \to \infty} \left(\frac{\sum_{c \in C} c(c)(\nu_c^l / l)}{1/l + \sum_{c \in C} r(c)(\nu_c^l / l)} \right) &= \frac{\lim_{l \to \infty} \left(\sum_{c \in C} c(c)(\nu_c^l / l) \right)}{\lim_{l \to \infty} \left(1/l + \sum_{c \in C} r(c)(\nu_c^l / l) \right)} \\
&= \frac{\sum_{c \in C} c(c) \lim_{l \to \infty} (\nu_c^l / l)}{\lim_{l \to \infty} (1/l) + \sum_{c \in C} r(c) \lim_{l \to \infty} (\nu_c^l / l)} \\
&\overset{\ddagger}{=} \frac{\sum_{c \in C} c(c) \pi(c)}{\lim_{l \to \infty} (1/l) + \sum_{c \in C} r(c) \pi(c)}
\end{aligned}$$

Equality ‡ holds because we have $\lim_{l \to \infty} \frac{\nu_c^l}{l} = \pi(c)$ by Lemma 4. The limit diverges to ∞ if and only if the rewards are all equal to zero and at least one cost is not. In this case the original definition of $\mathcal{R}$ diverges and hence $\mathcal{R}$ and the last expression are the same. Otherwise the last expression converges, so † converges, and so lim inf and lim of this sequence are the same. □

Note that the previous lemma implies that the value of a well-behaved run is independent of the actual run. In other words, on the set of well-behaved runs of a unichain Markov chain the ratio function is constant. So the expected value of such a Markov chain is equal to the ratio of any of its well-behaved runs.

Theorem 2 (Expected Ratio of a Unichain MC) *Let $\mathcal{C}$ be a unichain MC and let π denote the Cesaro limit of $p^n_{\mathcal{C}}$ of the induced Markov chain. Then the expected ratio can be calculated as follows.*

$$\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \frac{\sum_{c \in C} c(c)\pi(c)}{\lim_{l\to\infty}(1/l) + \sum_{c \in C} r(c)\pi(c)}$$

As a special case, when $r(c) = 1$ for all states, we can compute the mean payoff [34] as follows.

$$\mathbb{E}_{\mathcal{C}}[\mathcal{P}] = \sum_{c \in C} c(c)\pi(c)$$

Proof This follows from Lemma 6 and the fact that $\mathcal{R}$ is constant on a Markov chain (i.e., independent from the actual run). □

Note that this means that an expected value is ∞ if and only if the second cost of every action in the recurrence class of the Markov chain is 0 and there is at least one first cost that is not.

This provides us with an efficient method of calculating the expected ratio of a Unichain MC. We can calculate π by solving the linear equation system $\pi(P - I) = 0$ [34], where P is the probability matrix of $\mathcal{C}$ (Definition 4).

Each run of a MC will almost surely end in one recurrence class (the probability of visiting only transient states is equal to zero). And since $\mathcal{R}$ is prefix-independent, the ratio of this run will be equal to the ratio of the run inside the recurrence class.

Theorem 3 (Expected Ratio of a MC) *Let $\mathcal{C}$ be a MC. For each recurrence class $\mathcal{C}'$, let $\pi(\mathcal{C}')$ be the probability of reaching $\mathcal{C}'$.*

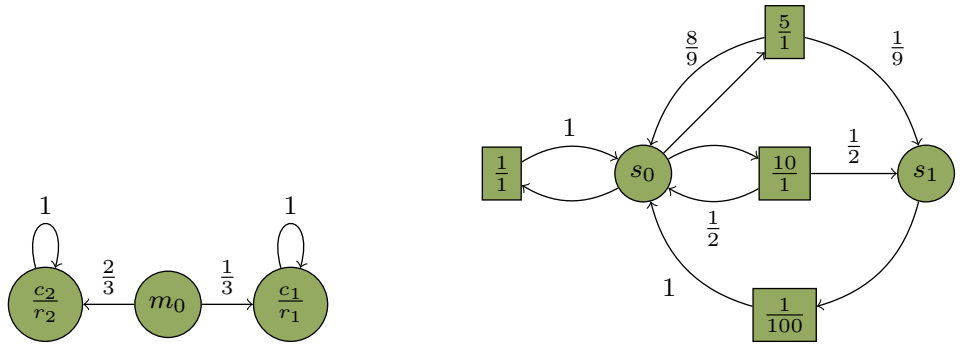
$$\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \sum_{\mathcal{C}' \, rec. \; class} \pi(\mathcal{C}')\mathbb{E}_{\mathcal{C}'}[\mathcal{R}],$$

where $\mathcal{C}'$ ranges over all recurrence classes of $\mathcal{C}$ and $\mathbb{E}_{\mathcal{C}'}[\mathcal{R}]$ denotes the expected ratio of the MC consisting only of recurrence class $\mathcal{C}'$.

3.4 Difference Between Ratio and Mean Payoff

Note that Theorem 3 also hints at the difference between the expected ratio and the ratio between expectations (as considered by Derman [14]). The following example shows that straight-forward reduction from MDPs with the ratio function to MDPs with the mean-payoff function is not possible.

Example 4 (Expected Ratio vs Ratio of Expectations) In Figure 3(a) we have a Markov chain with three states. m_0 is the initial state, and the states labelled with $\frac{c_1}{r_1}$ and $\frac{c_2}{r_2}$ are reached with probability $\frac{1}{3}$ and $\frac{2}{3}$, respectively. The labels also define the rewards and costs of each state.



(a) Expected Ratio vs Ratio of Expectations Example 4

(b) MDP for example Example 5

Fig. 3 Two examples showing that the Ratio objective cannot be easily reduced to the Mean objective

From the previous theorem it follows that we have $\mathbb{E}[\mathcal{R}] = \frac{1}{3} \cdot \frac{r_1}{c_1} + \frac{2}{3} \cdot \frac{r_2}{c_2}$. Note that this is not the same as dividing the expected average cost by the expected average reward $\frac{\mathbb{E}[\mathcal{P}_c]}{\mathbb{E}[\mathcal{P}_r]} = \frac{\frac{1}{3} \cdot c_1 + \frac{2}{3} \cdot c_2}{\frac{1}{3} \cdot r_1 + \frac{2}{3} \cdot r_2}$ (i.e., the ratio of expected average rewards and costs) for appropriate r_1, r_2, c_1 and c_2.

It is also not possible to just subtract costs from rewards and obtain the same result. Recall the automatic gear-shifting unit (ACTS) from Section 1, page 3. We want to optimize the relation of two measures: speed (km/h) and fuel consumption (l/h). When subtracting kilometers per hour from liters, the value of the optimal controller has no intuitive meaning. Furthermore, it can lead to non-optimal strategies, as shown by the following example.

Example 5 (Subtraction Leads to Different Strategies) Consider an MDP with two states, s_0 and s_1, as depicted in Figure 3(b). There is one action enabled in s_1. It has cost 1 and reward 100 and leads with probability 1 to s_0. There are three actions in s_0: Action a_0 has cost 5 and reward 1 and leads with probability 1/9 to s_1 and with 8/9 back to s_0. Action a_1 has cost 10 and reward 1 and leads with probability 1/2 to s_1 and with 1/2 to s_0. Action a_2 has cost and reward 1 and leads with probability 1 back to s_0. We will ignore this action for the remainder of this example.

The steady state distribution of the strategy choosing a_0 is $(9/10, 1/10)$, and so its ratio value is $(9/10 \cdot 5 + 1/10 \cdot 1)/(9/10 \cdot 1 + 1/10 \cdot 100) \approx 0.42$. For the strategy choosing a_1, the steady state distribution is $(2/3, 1/3)$ and the ratio value is $(2/3 \cdot 10 + 1/3 \cdot 1)/(2/3 \cdot 1 + 1/3 \cdot 100) \approx 0.634$, which is larger than for a_0. Hence choosing a_0 is the better strategy for the ratio objective (as we aim to minimize the ratio). If we now subtract the reward from the cost and interpret the result as a mean-payoff MDP, then we get rewards 4, 9, and -99 respectively. Choosing strategy a_0 gives us $9/10 \cdot 4 - 1/10 \cdot 99 = -6.3$, while choosing strategy a_1 gives us $2/3 \cdot 9 - 1/3 \cdot 99 = -27$. So, choosing a_1 is the better strategy for this objective.

Examples 4 and 5 show we cannot easily reduce the ratio payoff to mean payoff.

4 Algorithms

In this section we discuss three algorithms to calculate the most efficient strategies for MDPs. In all of them, we first decompose the MDPs into strongly connected components (called end components) and then calculate optimal strategies for each component. Finally we compose the resulting strategies into one optimal strategy for the complete MDP.

We will first discuss end components. Then we will define the common structure for all algorithms. Afterwards we will discuss three ways to compute optimal strategies for end components. Finally, we will evaluate the performances of all three algorithms and discuss their implications.

4.1 End Components

End components [13] of an MDP are defined as follows.

Definition 14 (End Component) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP. A subset of its states $M' \subseteq M$ is called an *end component* if

- for each pair of states $m, m' \in M'$ there is a strategy such that a run starting at m will reach m' with probability greater zero, and
- for each state $m \in M'$ there is an action $a \in A$ such that for all states $m' \in M$ with $p(m, a, m') > 0$ we have $m' \in M'$.

An end component is called *maximal* if there is no other end component that contains all its states.

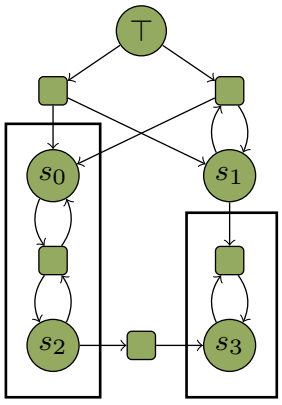


Fig. 4 end components

Figure 4 illustrates an MDP with two end components (inside the boxes)). The left end component consists of two states: s_0 and s_2. s_0 only has one possible choice: it has to go to the action below it, from which the next state is chosen probabilistically. However, s_2 has two possible choices: it can go up to the same action, or go right, from which the next state will be s_3. So both states in this end component can reach each other with probability one. Both states have an action to stay inside the end component. However, s_2 does not have to. There also is a strategy allowing it to only pass through this end component, instead of remaining in it. So, while every run has to end in an end component, a run that enters an end component does not have to stay there. Note further that state s_1 is not contained in an end component, although it can reach itself by picking the action above it. While there is an action from s_1 that has a path leading back to s_1, there is no strategy that enforces such a visit.

Lemma 7 (End Components Allow Optimal Unichain Strategies) *Let $\mathcal{M}$ be an end component, and let d be a non-unichain strategy for $\mathcal{M}$. Then there is a unichain strategy d' with expected ratio that is at least as good as that of d.*

```
   Input: MDP M, start state o_0
   Output: Value E[R] and optimal strategy d
1  ecSet ← decompose(M);
2  foreach i ← [0 ... |ecSet| − 1] do
3  |  switch ecSet_i do
4  |  |  case isZero :  λ_i ← 0; d_i ← zero-cost strategy ;
5  |  |  case isInfty :  λ_i ← ∞; d_i ← arbitrary ;
6  |  |  otherwise  : d_i ← solveEC(ecSet_i) ;
7  |  endsw
8  end
9  d ← compose(M, λ_0, ..., λ_{|ecSet|−1}, d_0, ..., d_{|ecSet|−1});
```

Algorithm 1: Finding optimal strategies for MDPs

```
   Input: MDP M, start state o_0
   Output: Set L of maximal end components
1  L ← {M};
2  while L cannot be changed anymore do
3  |  M' ← some element of L;
4  |  Deactivate all actions that lead outside of M';
5  |  Let M_1, ..., M_n be the strongly connected components of M';
   |  L ← L \ {M'} ∪ {M_1, ..., M_n};
6  end
```

Algorithm 2: Decomposition into maximal end components

Proof Lemma 3 and Definition 14 allow us to construct a unchain strategy from an arbitrary pure strategy with the same or a better value: d' fixes the recurrent class C' with the minimal value induced by d; for states outside of C', d' plays a strategy to reach C' with probability 1. □

4.2 General Algorithm Structure

As Lemma 7 shows, we can look for unichain strategies in the end components and then compose these into an optimal strategy for the whole MDP. The general shape of the algorithms is shown in Algorithm 1. In Line 1 we decompose the MDP into maximal end components. Then we analyze each end component separately: the predicates `isZero` and `isInfty` (Line 4 and 5, resp.) check if an end component has value zero or infinity. This is necessary because the algorithms calculating optimal strategies for end components (`solveEC`, Line 6) only work if a strategy with finite ratio exists and if the optimal strategy has ratio greater than zero. Finally, function `compose` (Line 9) takes values and strategies from all end components and computes an optimal strategy for $\mathcal{M}$ using Lemma 10.

Decomposition into end components, using the algorithm proposed in [13], happens in a sequence of refinements of MDPs, until no further refinement is possible. We describe this formally in Algorithm 2. Functions `isZero` and `isInfty` can be implemented efficiently as follows.

Lemma 8 *For every MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ such that M is an end component of $\mathcal{M}$, we can check efficiently if the value of $\mathcal{M}$ is zero or infinity and construct corresponding strategies.*

Proof $\mathcal{M}$ has value zero if there exists a strategy such that the expected average reward w.r.t. the cost function c is zero. We check this by removing all actions

from states in $\mathcal{M}$ that have $c > 0$ and then recursively removing all actions that lead to a state without enabled actions. If the resulting MDP $\mathcal{M}'$ is non-empty, then there is a strategy with value 0 for the original end component. It can be computed by building a strategy that moves to and stays in $\mathcal{M}'$.

$\mathcal{M}$ has value infinity iff (i) for every strategy the expected average reward w.r.t. cost function c is not zero, i.e., $\mathcal{M}$ has not value zero, and (ii) for all strategies the expected average reward w.r.t. the reward function r is zero. This can only be the case if for all actions in the end component the value of cost function r is zero. In this case, any arbitrary strategy will give value infinity. □

4.3 Algorithms for End Components

We will now discuss three algorithms for end components. For all of them, we assume that there exists a strategy with a finite ratio value and that the optimal strategy does not have value zero. The first two solutions are based on reduction to linear programs. The last solution is a new algorithm based on strategy (or policy) iteration.

Fractional Linear Program

Using Theorem 3, we transform the MDP into a fractional linear program. This is done in the same way as is done for the expected average payoff case (cf. [34]). We define variables $x(m, a)$ for every state $m \in M$ and every available action $a \in \overline{A}(m)$. This variable intuitively corresponds to the probability of being in state m and choosing action a at any time. Then we have for example $\pi(m) = \sum_{a \in \overline{A}(m)} x(m, a)$.

We need to restrict this set of variables. First of all, we always have to be in some state and choose some action, i.e., the sum over all $x(m, a)$ has to be one. The second set of restrictions ensures that we have a steady state distribution, i.e., the sum of the probabilities of going out of (i.e., being in) a state is equal to the sum of the probabilities of moving into this state.

Definition 15 (Fractional Linear program for MDP) Let $\mathcal{M}$ be a unichain MDP such that every Markov chain induced by any strategy contains at least one non-zero reward. Then we define the following fractional linear program for it.

$$\text{Minimize } f = \frac{\sum_{m \in M} \sum_{a \in \overline{A}(m)} x(m, a) \cdot c(m, a)}{\sum_{m \in M} \sum_{a \in \overline{A}(m)} x(m, a) \cdot r(m, a)} \tag{1}$$

$$\text{subject to } \sum_{m \in M} \sum_{a \in \overline{A}(M)} x(m, a) = 1 \text{ and} \tag{2}$$

$$\forall m \in M : \sum_{a \in \overline{A}(m)} x(m, a) = \sum_{m' \in M} \sum_{a \in \overline{A}(m')} x(m', a) \cdot p(m', a, m) \tag{3}$$

There is a correspondence between pure strategies and basic feasible solutions to the linear program[4]. That is, the linear program always has a solution because every positional strategy corresponds to a solution. See [34] for a detailed analysis of this in the expected average reward case that also applies here.

[4] A feasible solution is an assignment that fulfills the linear equations

Input: feasible solution x_0, MDP $\mathcal{M}$
Output: Variable assignment, optimal solution
$n \leftarrow 0$
repeat
  $g \leftarrow f(x_n)$; /* $f(x_n)$ denotes the value of Eqn. 1 under assignment x_n */
  $n \leftarrow n + 1$
  Minimize

$$\sum_{m \in M} \sum_{a \in \overline{A}(m)} x_n(m,a) \cdot c(m,a) - g \sum_{m \in M} \sum_{a \in \overline{A}(m)} x_n(m,a) \cdot r(m,a)$$

  subject to Equation 2 and Equation 3.
until $f(x_{n-1}) = f(x_n)$;
return x_n, $f(x_n)$

Algorithm 3: Reduction of fractional linear program to a sequence of linear programs [21]

In our implementation, we solve the fractional linear program in Definition 15 by solving a sequence of linear programs shown in Algorithm 3. This algorithm is due to [21]. Once we have calculated a solution of the linear program, we can calculate the strategy as follows.

Definition 16 (Strategy from solution of linear program) Let $x(m,a)$ be the solutions to the linear program. Let $M' = \{m \in M \mid \exists a \in A : x(m,a) > 0\}$. Then we define strategy d as $d(m) = a$ for all states $m \in M$ and the only possible $a \in A$ such that $x(m,a) > 0$. For all other states, choose a strategy such that M' is reached with probability 1 (Lemma 7).

Note that this is well defined because for each state m there is at most one action a such that $x(m,a) > 0$ because of the bijection (modulo the actions of transient states) between basic feasible solutions and strategies and because the optimal strategy is always pure and memoryless.

Linear Program

We can also use the following linear program proposed in [13] to calculate an optimal strategy. We are presenting it here for comparison to the other solutions later in this section.

Definition 17 (Linear program for MDP) Let $\mathcal{M}$ be a unichain MDP such that every Markov chain induced by any strategy contains at least one non-zero reward. Then we define the following linear program for it.

Minimize λ subject to:

$$b_m \leq c(m,a) - \lambda \cdot r(m,a) + \sum_{m' \in M} p(m,a,m') \cdot b_{m'} \qquad \forall m \in M, a \in \overline{A}(m) \qquad (4)$$

To calculate a strategy from a solution b_m to the LP, we choose the actions for the states such that the constraints are fulfilled when we interpret them as equations.

Input: MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$, mean payoff function $r : M \times A \to \mathbb{R}$
Output: Value $\mathbb{E}_{\mathcal{M}_d}[\mathcal{P}]$ and optimal strategy d

1 $n \leftarrow 0$, $d_0 \leftarrow$ arbitrary strategy;
2 **repeat**
3 Obtain vectors g_n, b_n that satisfy: $\begin{aligned} (P_{d_n} - I)g_n &= 0 \\ r_{d_n} - g_n + (P_{d_n} - I)b_n &= 0 \\ P^*_{d_n} b_n &= 0 \end{aligned}$;
4 $\overline{A}'(m) \leftarrow \arg\min_{a \in \overline{A}(m)} \sum_{m' \in M} p(m, a, m') g_n(m')$;
5 Choose d_{n+1} such that $d_{n+1}(m) \in \overline{A}'(m)$;
6 **foreach** $m \in M$ **do**
7 **if** $d_n(m) \in \overline{A}'(m)$ **then** $d_{n+1}(m) \leftarrow d_n(m)$;
8 **end**
9 **if** $d_n = d_{n+1}$ **then**
10 $\overline{A}'(m) \leftarrow \arg\min_{a \in \overline{A}(m)} [r(m) + \sum_{m' \in M} p(m, a, m') b_n(m')]$;
11 Choose d_{n+1} such that $d_{n+1}(m) \in \overline{A}'(m)$;
12 **foreach** $m \in M$ **do**
13 **if** $d_n(m) \in \overline{A}'(m)$ **then** $d_{n+1}(m) \leftarrow d_n(m)$;
14 **end**
15 **end**
16 $n \leftarrow n + 1$;
17 **until** $d_{n-1} = d_n$;

Algorithm 4: Finding optimal strategies for MDPs with mean payoff [34]

Policy Iteration

We will now design a policy iteration algorithm for $\mathcal{R}$ that is based on the policy iteration algorithm for mean cost $\mathcal{P}$, which we show in Algorithm 4. Recall that we use functions with finite domain and vectors interchangeably.

The goal of this algorithm is to find a strategy with minimal expected mean payoff. The algorithm consists of one loop in which we produce a sequence of strategies until no further improvement is possible (Line 17), i.e., until there is no strategy with smaller expected payoff. At the beginning of the loop we solve a linear equation system (Line 3). In this system, we denote by P_d the probability matrix we obtain from combining $\mathcal{M}$ with d, i.e., $P_d(m, m') = p(m, d(m), m')$. Analogously, r_{d_n} denotes the reward vector induced by strategy d_n, i.e., $r_{d_n}(m_i) = r(m_i, d_n(m_i))$. Finally, by I we denote an identity matrix of appropriate size. The resulting vectors are gain g and bias b. Gain $g(m)$ is equal to the expected payoff of a run starting in m. The bias can be interpreted as the expected total difference between a reward obtained in a state and the expected reward of that state [34]. Its detailed semantics is not of material importance to this paper.

In Line 4 we collect all possible actions for each state that minimize the local expected gain. In Line 5 we choose one strategy from the possible actions. To guarantee termination of this algorithm we fix the chosen strategy (Line 8) such that we choose the same action as the old strategy whenever possible. If it was not possible to find an improved strategy in this way, then we perform the steps from Line 4 to Line 8 with a different local target function (Line 9 to Line 14), based on reward and bias.

Theorem 4 (Algorithm 4 terminates and is correct) *Algorithm 4 always terminates and returns an optimal strategy.*

Proof In each iteration of the algorithm we have one of the following cases [34]

- $d_n = d_{n+1}$: In this case there is no better strategy.
- $d_n \neq d_{n+1}$: We know that either $g_n < g_{n+1}$ or $g_n = g_{n+1}$ and $b_n < b_{n+1}$ (i.e., we have a lexicographic ordering).

In the first case we know that we found the best possible strategy. This implies the correctness. From the second case it follows that no two strategies can show up twice except for the first case. Since there are only finitely many strategies we know that the algorithm therefore terminates. □

We are now going to reduce the search for an optimal ratio strategy for an MDP $\mathcal{M}$ with reward r and cost c to the search for an optimal mean cost strategy. According to Theorem 2, the ratio value of a unichain strategy d is $\lambda_d = \pi_d c_d / \pi_d r_d$, if we interpret c_d, r_d and π_d as vectors. Equivalently, $(c_d - \lambda_d r_d)\pi_d = 0$. If we now construct a mean payoff reward function $r' = c - \lambda r$, then d has therefore an expected mean payoff of zero. We call r' the *reward induced by* λ.

Definition 18 (Reward induced by λ) Let c and r be cost and reward functions and let $\lambda \in \mathbb{R}$ be a constant. Then we define the *reward induced by* λ as $r'(m,a) = c(m,a) - \lambda r(m,a)$.

The correlation between functions r, c and r' go even further, as the following lemma shows.

Lemma 9 (Relation of Ratio and Mean Payoff) *Let $\mathcal{M}$ be an MDP, r and c be payoff functions, d and d' be two unichain strategies with expected ratio λ and λ', respectively, and r' be the reward function induced by λ. Then the following all hold:*

1. *$\lambda' = \lambda$ if and only if the expected mean cost of d' in $\mathcal{M}$ with r' is zero i.e., $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] = 0$.*
2. *$\lambda' < \lambda$ if and only if the value of d' in $\mathcal{M}$ with r' is smaller than zero, i.e., $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] < 0 \iff \mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{R}_{\frac{c}{r}}] < \mathbb{E}_{\mathcal{M}_{d}}[\mathcal{R}_{\frac{c}{r}}]$.*
3. *If λ is not optimal, there is a strategy with value smaller than zero for $\mathcal{M}$ and r'.*

Proof For *1.*, $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] = 0$ if and only if $r'_{d'}\pi_{d'} = 0$ according to Theorem 2. By definition of r', this is equivalent to $(c_{d'} - \lambda r_{d'})\pi_{d'} = 0$. By vector arithmetic, this is equivalent to $c_{d'}\pi_{d'}/r_{d'}\pi_{d'} = \lambda$. According to Theorem 2, $c_{d'}\pi_{d'}/r_{d'}\pi_{d'} = \lambda'$. So we obtain $\lambda = \lambda'$.

For *2.*, assume that d' with r' has a value smaller than zero, i.e., $0 > \pi_{d'} r'_{d'} = \pi(c_{d'} - \lambda r_{d'})$ by the first claim, where $\pi_{d'}$ is the steady state distribution of d' in $\mathcal{M}$. Equivalently, $0 > \pi_{d'} c_{d'} - \pi_{d'} \lambda r_{d'}$ and $\lambda > \pi_{d'} c_{d'} / \pi_{d'} r_{d'} = \lambda'$, where the last equality follows from Theorem 3. Since all transformations are equivalent, the proof of this claim is finished.

For *3.*, assume that d^* is optimal in $\mathcal{M}$ with c and r and that its value is λ^*. Also assume that d, i.e., that $\lambda > \lambda^*$ is not optimal. We will now show that d^* has a value smaller than zero in the combination of $\mathcal{M}$ and r', i.e., prove the claim.

Let $v = \pi_{d^*} r'_{d^*} = \pi_{d^*}(c_{d^*} - \lambda r_{d^*})$ be the expected mean payoff value of d^* for r' and let $v^* = \pi_{d^*}(c_{d^*} - \lambda^* r_{d^*})$ be analogous for λ^*. From $\lambda^* < \lambda$ it follows that $v^* > v$. Since v^* is the value of d^* in the combination of $\mathcal{M}$ and the reward induced

Input: End Component $\mathcal{M}$, unichain strategy d_0 (with $0 < \lambda_0 < \infty$)
Output: Optimal unichain strategy d_n
1 $n \leftarrow 0$;
2 **repeat**
3 $\quad \lambda_n \leftarrow \mathbb{E}_{\mathcal{M}_{d_n}}[\mathcal{R}]$;
4 $\quad d_{n+1} \leftarrow$ improved unichain strategy for $\mathcal{M}_{\lambda_n}$;
5 $\quad n \leftarrow n + 1$;
6 **until** $d_{n-1} = d_n$;

Algorithm 5: Policy iteration for $\mathcal{R}$

by λ^*, we know that v^* is zero from the first claim. From $0 = v^* > v$ we have that v is smaller than zero. But v is the value of d^* in the combination of $\mathcal{M}$ and r' by definition, i.e., d^* is a better strategy in the induced MDP. □

Lemma 9 allows us to find an optimal strategy for $\mathcal{R}$ by starting with some strategy d with value $\lambda < \infty$. We can then look for a better strategy with Algorithm 4 in the combination of $\mathcal{M}$ and the function induced by λ. If we cannot find such a strategy, then d is optimal, according to the third claim of the last lemma. If we find such a strategy, then it has a ratio lower than λ according to the second claim of the last lemma. This leads us to Algorithm 5.

This algorithm is correct and terminates since the expected values we produce are always decreasing. From Lemma 9 it follows that the algorithm will always find a correct strategy. Note that it is undefined how far we improve the strategy in Line 4. We can take the first strategy having an expected payoff smaller than zero or we can find an optimal strategy. As we will see in Section 4.5 there seems to be little difference between the two approaches. However, the following example shows that choosing the best strategy in the induced MDP is not always beneficial.

Example 6 (Choosing Optimally in the Induced MDP is not Always Optimal) Consider Figure 3(b) on Page 16. If we choose for state s_0 the action with cost 1 and reward 1, then we obtain 1 as expected ratio payoff of this MDP. In the MDP induced by 1 we have -6.3 as expected mean payoff for choosing the action with cost 5 and reward 1, according to Example 5. Analogously, we have -27 for choosing the other action. So, if we choose the optimal strategy in the induced MDP we will pick the latter action. But, as seen in Example 5, the former is optimal. □

Theorem 5 (Correctness of Algorithm 5) *Algorithm 5 terminates and is correct.*

Proof Two strategies with different efficiencies cannot be the same. The ratios in Algorithm 5 are monotonically improving. There are only finitely many strategies. So termination follows. Correctness follows from Lemma 9. □

4.4 Composing MDPs

Once we have calculated optimal strategies for end components, we calculate a strategy that selects end components and decisions to reach them optimally. To that end, we employ algorithms for calculating optimal strategies of mean-payoff MDPs, as discussed above.

We construct a new MDP in which each end component is represented by one state, and each state not in an end component is represented by itself. If it was

possible to move from one state or end component to another with a given action, then it will be possible to move from one representing state to the other in the new MDP. We will assign rewards such that staying in a state representing an end component is rewarded by the expected payoff of that component. Moving from one component to another has no cost. An optimal strategy for this MDP defines an optimal strategy for states not in an end component as well as movement between end components.

Lemma 10 *Given an MDP $\mathcal{M}$ and an optimal pure strategy d_i for every maximal end component $C_i, 1 \leq i \leq n$ in $\mathcal{M}$, we can compute the optimal value and construct an optimal strategy for $\mathcal{M}$.*

Proof Let λ_i be the value obtained with d_i in the MDP induced by C_i. Without loss of generality, we assume that every action is enabled in exactly one state.

Let $\mathcal{M}' = (M', m_0', A', \overline{A}', p')$ be an MDP constructed from $\mathcal{M}$ as follows:

- $M' = \{C_i \mid \forall 1 \leq i \leq n\} \cup \{m \in M \mid m \notin \bigcup C_i\}$,
- $m_0' = \begin{cases} C_i & m_0 \in C_i \\ m_0 & \text{otherwise} \end{cases}$
- $A' = A$,
- $\overline{A}' = \{(m, a) \in \overline{A} \mid m \notin \bigcup C_i\} \cup \{(C_i, a) \mid \exists m \in C_i \wedge (m, a) \in \overline{A}, 1 \leq i \leq n\}$, and
- let p' be such that it respects the following conditions:
  - $\forall m, m' \in M \cap M' \; \forall a \in A : p'(m, a, m') = p(m, a, m')$, i.e., probability of moving between two states m and m' that are outside of any end component are the same as in $\mathcal{M}$,
  - $\forall m \in M \cap M' \; \forall a \in A \; \forall 1 \leq i \leq n : p'(m, C_i) = \sum_{m' \in C_i} p(m, a, m')$, i.e., the probability of moving from a state m outside of any end component to end component C_i in $\mathcal{M}'$ is equal to the probability of moving from m to any of the states of C_i in $\mathcal{M}$,
  - $\forall m \in M \cap M' \; \forall a \in A \; \forall 1 \leq i \leq n : p'(C_i, a, m) = \max_{m' \in C_i} p(m', a, m)$, i.e., the probability of moving from end component C_i to a state m that is outside of any end component with action a is equal to the probability of moving from the single state a' in which a is activated to m,
  - $\forall a \in A \; \forall 1 \leq i, j \leq n : p'(C_i, a, C_j) = \max_{m \in C_i} \sum_{m' \in C_j} p(m, a, m')$, i.e., the probability of moving from end component C_i to end component C_j with action a is equal to the probability of moving from the single state in which a is activated to any of the states in C_j; it is zero if no such state exists.

We modify $\mathcal{M}'$ to obtain an MDP $\mathcal{M}''$ by removing all actions for which there is a state $m \in M'$ such that $p'(m, a, m) = 1$. Furthermore, for all states C that represent an end component in $\mathcal{M}$ with value $\lambda_i < \infty$, we add a new action a_i with $p(C, a_i, C) = 1$ and costs λ_i; all other actions have cost 0. We now recursively remove states without enabled actions and actions leading to removed states. If the initial state m_0 is removed, the MDP has value infinity, because we cannot avoid reaching and staying in an end component with value infinity.

Otherwise, let d'' be an optimal strategy for $\mathcal{M}''$. We define d by $d(m) = d''(m')$ for all states $m \notin \bigcup C_i$. For $m \in C_i$, if $d''(m') = a_i$, we set $d(m) = d_i(m)$. Otherwise, let a be the actions chosen in state m', and let m'' be the state in which a is enabled. Then, we set $d(m'') = a$ and for all other states in C_i we choose d such that we reach m'' with probability 1. We can choose the strategy arbitrarily in states that

were removed from $\mathcal{M}''$, because these states will never be reached by construction of d. Because of the way we constructed $\mathcal{M}''$, d and d'' have the same value, and d is optimal because d'' is optimal (Theorem 3). □

4.5 Evaluation

The goal of this first evaluation is to find out which of the given implementations we should follow to try to scale to large systems. We therefore apply all three implementations to a series of production line configurations of increasing size. We also report on the synthesized strategies.

Test Cases and Results.

We synthesized optimal controllers for systems with two to five production lines, i.e., the underlying MDP is a product of two to five copies of the environment model (shown in Figure 1(a)) and the monitor (shown in Figure 1(b)).

The synthesized controllers behave as follows: For a system with two production lines, the controller plays it safe. It turns one production line on in fast mode and leaves the other one turned off. If the production line breaks, then the other production line is turned on in slow mode and the first production line is repaired immediately. For three production lines, all three production lines are turned on in fast mode. As soon as one production line breaks, only one production line is turned on in fast mode, the other one is turned off. Using this strategy, the controller avoids the penalty of having no working production line with high probability. If two production lines are broken, then the last one is turned on in fast mode and the other two production lines are been repaired. In the case of four production lines, all production lines are turned on in fast mode if they are all working. If one production line breaks, then two production lines are turned on and the third working production line is turned off. The controller has one production line in reserve for the case that both used production lines break. If two production lines are broken, then only one production line is turned on, and the other one is kept in reserve. Only if three production lines are broken, the controller starts repairing the production lines. Using this strategy, the controller maximizes the discount for repairing multiple production lines simultaneously.

We also evaluated the ACTS described in Section 1. The model has two parts: a motor (the system model) and a driver profile (environment model). The state of the motor consists of revolutions per minute (RPM) and a gear. The RPM ranges from 1000 to 6000, modelled as a number in the interval (10, 60), and we have three gears. The driver is meant to be a city driver, i.e., who changes between acceleration and deceleration frequently. Fuel consumption is calculated as a polynomial function of degree three with the saddle point at 1800 rpm. The final model has 384 states, which takes less than a second to build. Finding the optimal strategy takes less than a second. The resulting expected fuel consumption is 0.15 l/km. The optimal strategy is as expected: shifts occur as early as possible.

Table 1 Experimental results table

n	$\|M\|$	$\|\overline{A}\|$	LP		FLP		Opt		Imp.	
2	9	144	0.002	13	0.015	14	0.003	13	0.003	14
3	27	1728	0.043	14	0.642	20	0.027	13	0.009	14
4	81	20736	1.836	41	14.73	332	0.122	21	0.122	24
5	243	248832	67.77	505	n/a	n/a	1.647	162	1.377	166

Experiments.

We have implemented the algorithms presented in this paper. Our first implementation is written in `Haskell`[5] and consists of 1500 lines of code. We use the Haskell package `hmatrix`[6] to solve the linear equation system and `glpk-hs`[7] to solve the linear programming problems. In order to make our work publicly available in a widely used tool and to have access to more case studies, we have implemented the best-performing algorithm within the explicit-state engine of PRISM. It is an implementation of the strategy improvement algorithm and uses numeric approximations instead of solving the linear equation systems.

First, we will give mean running times of our `Haskell` implementation on the production line example, where we scale the number of production lines. The tests were done on a Quad-Xeon with 2.67GHz and 3GB of heap space. Table 1 shows our results. Column n denotes the number of production lines we use, $|M|$ and $|\overline{A}|$ denote the number of states and actions the final MDP has. Note that $|M| = 3^n$ and $|\overline{A}| = 12^n$. The next columns contain the time (in seconds) and the amount of memory (in MB) the different algorithms used. LP denotes the linear program, FLP the fractional linear program. We have two versions of the policy iteration algorithm: one in which we improve the induced MDP to optimality (Column Opt.), and one where we only look for any improved strategy (Column Imp.). The policy iteration algorithms perform best, and Imp. is slightly faster than Opt but uses a little more memory. For $n = 5$, the results start to differ drastically. FLP ran out of memory, LP needed about a minute to solve the problem, and both Imp. and Opt. stay below two seconds.

Using our second implementation, we also tried our algorithm on some of the standard PRISM benchmarks [25]. For example, we used the IPv4 zeroconf protocol model. We asked for the minimal expected number of occurrences of action `send` divided by occurrences of action `time`. If we choose $K = 5$ and `reset = true`, then the resulting model has 1097 states and finding the optimal strategy takes 5 seconds. For $K = 2$ and `reset = false`, the model has about 90000 states and finding the best strategy takes 4 minutes on a 2.4GHz Core2Duo P8600 laptop.

5 Symbolic Implementation

In this section, we will discuss a symbolic variant of the policy iteration algorithm, i.e., the structure of Algorithm 1 with Algorithm 5 implementing `solveEC`. Symbolic encoding via *binary decision diagrams* (*BDDs*) has enabled model checking

[5] `http://www.haskell.org`

[6] `http://code.haskell.org/hmatrix/`

[7] `http://hackage.haskell.org/package/glpk-hs`

and qualitative synthesis to address the state explosion problem in many cases [9]. An extension called *multi-terminal BDDs* (*MTBDDs*) has also been widely used for symbolic implementations of probabilistic model checking, and is a core part of the PRISM tool, upon which we develop our techniques.

Recently, Wimmer et al. developed a semi-symbolic (or, in their terms, *symblicit*) variant of Algorithm 4 [39] with promising results. In this section, we develop an analogous algorithm for the case of Ratio-MDPs. We call the algorithm semi-symbolic because it uses symbolically as well as explicitly encoded MDPs. BDDs are good for encoding large structures but they are not suitable when it comes to solving linear equation systems (see for example [19,23]). Therefore we (like [39]) encode MDPs and strategies symbolically but convert the induced MC into an explicit linear equation system. For efficiency, before conversion, we reduce the MC using symbolic bisimulation minimization [40].

In what follows, we give an overview of the symbolic implementation. We first recall how to encode MDPs using MTBDDs and then describe how all the key parts of Algorithm 1 are implemented using BDDs and MTBDDs.

5.1 Symbolic Encoding

BDDs encode Boolean functions $2^{\mathrm{V}} \to \mathbb{B}$, where V is a finite set of variables, as directed acyclic graphs [8]. Given, for example, $\mathrm{V} = \{v, w\}$ and function $f : 2^{\mathrm{V}} \to \mathbb{B}$, we write $f(\{v\})$ to denote the function value of the variable assignment $v = 1 \wedge w = 0$. Accordingly, $f(\emptyset)$ denotes the value of f for the assignment $v = 0 \wedge w = 0$.

BDDs support several logical operations very efficiently (linear in the number of nodes). Given functions stored as BDDs $\mathcal{B}$ and $\mathcal{B}'$, and variables $\mathrm{V}' \subseteq \mathrm{V}$, we can easily perform: negation ($\neg\mathcal{B}$), conjunction ($\mathcal{B} \wedge \mathcal{B}'$), disjunction ($\mathcal{B} \vee \mathcal{B}'$), existential quantification ($\exists\, \mathrm{V}' : \mathcal{B}$) and universal quantification ($\forall\, \mathrm{V}' : \mathcal{B}$).

MTBDDs [17,1] are an extension of BDDs that encode functions $2^{\mathrm{V}} \to \mathbb{R}$, i.e., real-valued rather than Boolean-valued functions. Again, many useful operations can be implemented efficiently. Given MTBDDs $\mathcal{B}$ and $\mathcal{B}'$, variables $\mathrm{V}' \subseteq \mathrm{V}$ and constant $c \in \mathbb{R}$, we can perform: negation ($-\mathcal{B}$), addition ($\mathcal{B} + \mathcal{B}'$), multiplication ($\mathcal{B} \times \mathcal{B}'$), division ($\mathcal{B} \,/\, \mathcal{B}'$), minimization ($\min_{\mathrm{V}'} : \mathcal{B}$), maximization ($\max_{\mathrm{V}'} : \mathcal{B}$), summation ($\sum_{\mathrm{V}'} : \mathcal{B}$) and comparison with a constant ($\mathcal{B} < c$).

MTBDDs can be used to represent and manipulate matrices [17,1] and probabilistic models such as MCs and MDPs [19,2,31]. Consider, for example, an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$. To represent this symbolically, we first need a symbolic encoding of the state space M using $\lceil \log_2(|M|) \rceil$ variables V_M, described as an injective function $\mathrm{enc}_M : M \to 2^{\mathrm{V}_M}$, and a symbolic encoding of A using $\lceil \log_2(|A|) \rceil$ variables V_A, described as an injective function $\mathrm{enc}_A : A \to 2^{\mathrm{V}_A}$.

We encode the action activation relation $\overline{A}$ of $\mathcal{M}$ as a BDD representing the function $f_{\overline{A}} : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{B}$ such that $f_{\overline{A}}(\mathrm{enc}_M(m) \cup \mathrm{enc}_A(a)) = 1$ if and only if $(m, a) \in \overline{A}$. To represent the transition function p of $\mathcal{M}$, we also need a second encoding of the state space $\mathrm{enc}'_M : M \to 2^{\mathrm{V}'_M}$, which is identical to enc_M, but mapping to a second set of $\lceil \log_2(|M|) \rceil$ variables V'_M. We encode p as an MTBDD representing the function $f_p : 2^{\mathrm{V}_M \cup \mathrm{V}_A \cup \mathrm{V}'_M} \to \mathbb{R}$ such that $f_p(\mathrm{enc}_M(m) \cup \mathrm{enc}_A(a) \cup \mathrm{enc}'_M(m')) = p(m, a, m')$ for all states $m, m' \in M$ and actions $a \in A$ and 0 for every other assignment.

```
   Input: S, p, E
   Output: L
1  L_0 ← [(S, E)];
2  n ← 0;
3  repeat
4  |  L_{n+1} ← [];
5  |  foreach (S', E') ← L_n do
6  |  |  Let (S_1, E_1), ..., (S_l, E_l) be the SCCs of (S', E') ;
7  |  |  E'_i ← E_i ∧ [∃a∀m' : p → S_i)] ;
8  |  |  Add (S_i, E'_i) to L_{n+1};
9  |  |  n ← n + 1;
10 |  end
11 until L_n = L_{n−1};
```

Algorithm 6: Symbolic End Component Computation

Other entities we may need are encoded similarly. A set of states $S \subseteq M$ is represented by a function $f_S : 2^{\mathrm{V}_M} \to \mathbb{B}$, a pure strategy $d : M \to A$ by a function $f_d : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{B}$, and costs and rewards as functions $f_c, f_r : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{R}$. MCs are encoded as for MDPs but omitting the actions.

5.2 Symbolic MDP Decomposition

We need to implement each part of Algorithm 1 symbolically, using the encoding described above. The initial phase of the algorithm is the decomposition of the MDP into its maximal end components, shown previously in Algorithm 2.

First, we generate a BDD E for the underlying graph of the MDP by replacing every leaf in p that has a value greater than 0 by 1. We further abstract existentially over all actions, i.e., $E = \exists_{\mathrm{V}_A}(p > 0)$. Notice that, for clarity of presentation, we use the same symbol for an (MT)BDD and what it represents (e.g., p is the MTBDD representing transition function p). We now have that E represents the function f_E where $f_E(\mathrm{enc}_M(m) \cup \mathrm{enc}'_M(m'))$ if and only if there is any action that makes it possible to move from state m to state m' in $\mathcal{M}$.

Algorithm 6 works in exactly the same way as Algorithm 2. In Line 6, we assume that there is a method of decomposing into strongly connected components symbolically (see e.g. [5]). The only important line is Line 7, where we restrict the set of possible actions to those that stay inside an EC. Here we restrict the directed graph such that there is only an edge if there is any possible action that stays inside the end component.

After identification of end components, we need to implement the checks `isZero` and `isInfty` from Algorithm 1, which is done with a symbolic version of what we described in Lemma 8. For `isZero`, we restrict the set of states to those that have a cost of zero, i.e., $M \wedge \exists_{\mathrm{V}'_M} \exists_{\mathrm{V}_A} : c = 0$. If the resulting MDP has an end component, then `isZero` will return true. For `isInfty` we check if $r = 0 \wedge [\neg M \vee (\exists_{\mathrm{V}_M} \exists_{\mathrm{V}_A} : c > 0)]$ is a tautology.

5.3 Symbolic Policy Iteration

The next part of Algorithm 1 is the optimization of each individual end component using policy iteration. In Algorithm 7 we present the algorithm that finds the

```
   Input: MDP mdp consisting of a single end component
   Output: strategy d and optimal ratio value λ of mdp
 1 d = initial(mdp) ;
 2 d_old = ⊥;
 3 while d ≠ d_old do
 4 |   λ = lambda(mdp, d);
 5 |   g, b = gainAndBias(mdp, d, λ);
 6 |   while g ≥ 0 and d_old ≠ d do
 7 |   |   d_old = d;
 8 |   |   d = next(mdp, d, λ, g, b);
 9 |   |   g, b = gainAndBias(mdp, d, λ);
10 |   end
11 end
12 return unichain(mdp, d), λ
```

Algorithm 7: Optimisation for a Single End Component

optimal ratio value for an end component. The algorithm first picks any strategy d that has a finite and strictly positive value (Line 1). We observed that the choice of this strategy has a strong influence on the performance of our algorithm.

Then, in Line 3 we enter a loop that produces in every iteration a new strategy that has the same or a better ratio value (λ) than the previous strategy. We exit the loop if the same strategy is produced twice, i.e., there is no strategy with a better ratio value for this MDP.

In the loop, we first compute the ratio value λ that can be obtained by a strategy generated from the strategy d (Line 4). This computation is done semi-symbolically. First, we compute the Markov chain C induced by strategy d. We then apply symbolic bisimulation minimization to C, which allows us to construct an equivalent smaller Markov chain C'. Then, we compute all recurrence classes (i.e., the strongly connected components) of C'. For each recurrence class, we build an explicit-state representation of the sub-model and calculate the steady-state distribution, which in turn is used to calculate the ratio value of the recurrence class. We set λ to the value of the best recurrence class. This value is not necessarily the value of d but we can construct a strategy that has value λ. Furthermore, λ is at least as good as the actual value of d (see proof of Lemma 7).

In the rest of the algorithm, we perform computation on an MDP with average objective induced by the reward function $c - \lambda \times r$ (which we compute symbolically). For this induced MDP, we compute gain (g) and bias (b) (as in Algorithm 4). The computation of gain and bias is similar to the computation in Algorithm 4, i.e., we calculate gain and bias explicitly on an equivalent smaller Markov chain. We know that a state has a gain smaller than zero in the induced MDP if and only if its ratio value is smaller (i.e., better) than the ratio value from which the induced MDP was calculated (Lemma 9). Since the ratio value of strategy d is at most as good as λ, the gain of all states at this point is greater than or equal to zero.

We now enter the inner loop (Line 6), which runs while the strategy keeps changing and all entries of the gain vector are greater than or equal to zero. Equivalently, the loop runs until there is a recurrence class of the current strategy that has a value smaller than λ or until there is no better strategy any more.

In the inner loop, we try to improve the strategy (Line 8) and calculate the new strategy's gain and bias (Line 9). Note that the choice of the next strategy and the way of computing the value λ differs from our description in Section 4. In

the latter, we demanded a unichain strategy from the induced Mean MDP; here, we allow any kind of strategy. Forcing the algorithm to use a unichain strategy was a major bottleneck in our initial symbolic implementation, because it introduced irregularity into the (MT)BDDs.

Instead we work with arbitrary strategies now. To calculate the expected ratio of a strategy, we calculate the expected ratio of each recurrent class with Theorem 2 and take their minimum. The correctness of this approach follows trivially from the existence of a unichain Strategy with the same recurrence class as the optimal recurrence class, and therefore with the same value Lemma 7.

5.4 Symbolic Composition

We can build a strategy for the whole MDP once we have built a strategy for the end components. This could be done exactly as in the explicit variant, and by using a symbolic policy algorithm for mean-payoff MDPs, i.e., with a symbolic variant of Algorithm 4. Instead, we reduce the problem of composing strategies to the problem of finding a *stochastic shortest path*.

Definition 19 (Stochastic Shortest Path Problem) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP, and $r : M \times A \to \mathbb{R}$ be a reward function. Then the *Total Reward* $\mathcal{F}_r : (M \times A)^\omega \to \mathbb{R}$ defined by r is defined as $\mathcal{F}_r(\rho) = \sum_{i=0}^{\infty} r(\rho_i)$. For a state $m \in M$, an optimal strategy for the stochastic shortest path problem is one for which the expected value is minimal, i.e., $\arg\min_{d \in D'} \mathbb{E}_{\mathcal{M}_d}[\mathcal{F}]$, where $D' \subseteq D(\mathcal{M})$ is the set of pure strategies reaching m almost surely.

We use the following definition to reduce strategy composition to a simple variant of the shortest stochastic path problem.

Definition 20 (Reduction to Stochastic Shortest Path Problem) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be a MDP, $r : M \to \mathbb{R} \cup \{\bot\}$ be the optimal ratio calculated for each state so far, or $\bot$ if no reward has been calculated (because the state does not lie in any end component).

We construct a new MDP $\mathcal{M}' = (M', m_0', A', \overline{A}', p')$ by adding a special state $m_\bot$ to the set of states and keeping the start state, i.e., $M' = M \cup \{\bot\}$, and $m_0' = m_0$. Let $E \subseteq M$ be the set of states in any end component of $\mathcal{M}$.

We extend the set of inputs by a fresh symbol $a_\bot$, i.e., $A' = A \cup \{a_\bot\}$, where $a_\bot$ is available in all states in any end component, i.e., $\overline{A}' = \overline{A} \cup E \times \{\bot\}$. For any input $a \in A$ in $\mathcal{M}$, the transition function p' of $\mathcal{M}'$ is the same as in $\mathcal{M}$, i.e., $p'(m, a, m') = p(m, a, m')$ for all states $m, m' \in M$. For the new input $a_\bot$ we define $p'(m, a_\bot, \bot) = 1$ for all $m \in E \cup \{\bot\}$.

As reward function r' we assign $r'(m, \bot) = r(m)$ for every state $m \in E$, and 0 for all other states and actions. (Note that all states in an end component have the same value, so $r(m)$ is the reward of the end component.)

The correctness of the above reduction follows from the fact that the expected ratio payoff of a strategy of an MDP is equal to the sum over all end components C, in which the expected ratio of C is multiplied by the probability of reaching and staying in C, i.e., $\mathbb{E}^d_{\mathcal{M}}[r] = \sum_C p(C)\mathbb{E}^d_C[r]$. So, whenever the strategy chooses the edge $(m, \bot)$, then it is optimal to stay in the end component of m. For every

Input: min. Reward r, symb. transition f_p, immediate reward I
Output: Values V', Strategy d
1 $V' = r$;
2 $V'(\bot) = 0$;
3 $V = \infty$;
4 **while** $||V - V'|| \geq \epsilon$ **do**
5 $\quad V = V'$;
6 $\quad V' = \min_{\mathrm{V}_p} \sum_{\mathrm{V}_M} \left(f_p \cdot V'(\mathrm{V}'_M \leftarrow \mathrm{V}_M) + I \right)$
7 **end**

Algorithm 8: Solving the Stochastic Shortest Path Problem

other state (i.e., transient states or end components that we want to traverse), an optimal strategy of the stochastic shortest path problem is the optimal strategy for the orignal (ratio-payoff) problem.

We use Value Iteration to solve this problem. This class of algorithms has been studied extensively, and we will therefore not dwell on its correctness.

Lemma 11 (Algorithm 8 terminates and is correct) *When given a symbolically encoded MDP with r the minimal reward optimal reward of all end components, f_p the symbolic transition function and immediate reward function $I : M \times A \to \mathbb{R}$, Algorithm 8 terminates and delivers an ϵ-optimal strategy and ϵ-optimal values, i.e., the value of its strategy is at most ϵ away from the optimal value in the $|| \cdot ||$-norm.*

Proof See [34].

Using Algorithm 8 and Definition 20, we obtain the following theorem.

Theorem 6 (Composing strategies) *Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP and let $\mathcal{M}'$ be constructed from $\mathcal{M}$ as in Definition 20. Let $\mathcal{E} \subseteq 2^{\mathcal{M}}$ be the set of end components of $\mathcal{M}$, d_E the optimal strategy for all end components $E \in \mathcal{E}$, and let d' be the optimal strategy for $\mathcal{M}'$. We call an end component E active if there is a state $m \in E$ such that $d'(m) = a_\bot$. Define $d(M) \in A$ for m as*

- *$d(m) = d_E(m)$ if there is an active end component $E \in \mathcal{E}$ such that $m \in E$*
- *$d(m) = d'(m)$ otherwise*

Then d is optimal for $\mathcal{M}$.

Proof First note that for every strategy d in $\mathcal{M}$ there is an analogous strategy d' in $\mathcal{M}'$, and vice versa. We define d based on d' as defined above. For the other direction, call and end component active if the probability of visiting and staying in this end component is greater 0. We define d' based on d by assigning $d'(m) = d(m)$ for all states m not in an active end component. For all active end components E and all states $m \in E$, we assign $d'(m) = a_\bot$. Finally, we assign $d'(\bot) = a_\bot$.

Then, for every optimal strategy d for $\mathcal{M}$, d' is optimal for $\mathcal{M}'$, and vice versa. To see why, note that $\mathbb{E}_{\mathcal{M}_d}[\mathcal{R}] = \sum_{E \in \mathcal{E}} p_d(E) r(E) = \sum_{E \in \mathcal{E}} p'_{d'}(E_\bot) r(E) = \mathbb{E}_{\mathcal{M}'_{d'}}[\mathcal{F}]$, where by $p_d(E)$ we denote the probability that a run in $\mathcal{M}$ will reach end component E and stay in it, given that strategy d is used. Analogously, by $p'_{d'}(E_\bot)$ we denote the probability that a run in $\mathcal{M}'$ will take action $a_\bot$ from a state in E, given that strategy d is used.

We chose this construction for the symbolic encoding because no numerical computations (i.e., no equation solving) is involved. This is a great asset when dealing with MTBDDs.

Table 2 Experimental results table

Name	#States	#Blocks	Time in sec	RAM in MB
*line*3	386	271	0.9	112
*line*4	1560	945	5.6	150
*line*5	5904	3089	20.5	236
*line*6	21394	9448	96.8	326
*rabin*3	27766	722	5.2	199
*rabin*4	668836	12165	104.6	537
zeroconf	89586	29427	2948.7	608
acts	1734	1734	1.6	159
*phil*6	917424	303	1.2	181
*phil*7	9043420	303	1.9	262
*phil*8	89144512	342	2.6	295
*phil*9	878732012	342	3.3	287
*phil*10	8662001936	389	4.3	303
*power*1	8904	72	0.415	89.9
*power*2	8904	n/a	n/a	85

5.5 Evaluation of the Symbolic Algorithm

Table 2 shows the results of our implementation on various benchmarks. The implementation can be downloaded from `http://www-verimag.imag.fr/~vonessen/ratio.html`. The first column shows the name of the example; column `#States` denotes the number of states the model has; `#Blocks` the maximum number of blocks into which states are partitioned by bisimulation minimization while analyzing the model; `Time` the total time needed; `RAM` the amount of memory used (including all memory used by PRISM and its Java Virtual Machine). Below, we briefly describe the examples and discuss the results.

Experiments. Examples *line*3-6 model the assembly line system described in Section 1. We optimize the ratio between maintenance costs and number of units produced by several lines running in parallel (recall that the underlying MDP is the product of three to six copies of the environment model shown in Figure 1(a) and three to six copies of the specification monitor shown in Figure 1(b)).[8] Example *zeroconf* is based on a model of the ZeroConf protocol [26]. We modify it to measure the best-case efficiency of the protocol, finding the expected time it takes to successfully acquire an IP address. We choose a model with two probes sent, two abstract clients and no reset. This model shows the limit of our technique when bisimulation produces many blocks. In experiments *phil*6-10, we use Lehmann's formulation of the dining philosophers problem [27]. Here we measure the amount of time a philosopher spends. This model is effectively a mean-payoff model because we have a cost of one for each step. We use this experiment to compare our implementation to [39]. We are several orders of magnitude faster. We attribute the increase in speed to good initial strategy. These experiments also show the effectiveness of bisimulation minimization on state spaces with a very regular structure. In *rabin*3 and *rabin*4, we measure the efficiency of Rabin's

[8] Note that, due to the way that we model the assembly line here in PRISM, these are different sized MDPs to the ones for the same example used in Table 1.

mutual exclusion protocol [35]. We minimize the time of a process waiting for its entry into the critical section per entry into the critical section. Note that only the ratio objective allows us to measure exactly this property, because we grant a reward every time a process enters the section and a cost for every time a process has to wait for its entry. We also modelled an automatic clutch and transmission system (*acts*). Each state consists of a driver/traffic state (waiting in front of a traffic light, breaking because of a slower car, free lane), current gear (1-4) and current motor speed (100 - 500 RPM). We modelled the change of driver state probabilistically, and assumed that the driver wants to reach a given speed (50 km/h). Given this driver and traffic profile, the transmission rates and the fuel consumption based on motor speed, we synthesized the best points to shift up or down. In *power*1-2, we used the example from [29,16], which the authors use to analyze dynamic power management strategies. Our implementation allows solution of optimization problems that are not possible with either [29] or the multi-objective techniques in [16]. For example, in *power*1 we ask the question "What is the best average power consumption per served request". In *power*2, we ask for the best-case power-consumption per battery lifetime, i.e., we ask for how many hours a battery can last.

Observations. The amount of time needed by the algorithms strongly depends on the number of blocks (from bisimulation minimization) that it constructs. We observed that a higher number of blocks increases the time necessary to construct the partition. Each minimization refinement step takes longer the more blocks we have. Analogously, the more blocks we have, the bigger the matrices we need to analyze. We observed an almost monotone increase in the number of blocks while policy iteration runs. Accordingly, it is beneficial to select an initial strategy with as few blocks as possible.

In the original policy iteration algorithm of Section 4, we constructed unichain strategies from Multichain strategies several times throughout the algorithm. As it turns out, unichain strategies increase the amount of blocks dramatically. We therefore successfully modified our algorithm to avoid them, which drastically improved performance.

The symbolic encoding as well as bisimulation are crucial to handle models of a size that the explicit implementation described in Section 4 could not handle (storing a model of the size of *phil10* was not feasible on our testing machine).

6 Conclusion

We have presented a framework for synthesizing efficient controllers, based on finding optimal strategies in Ratio-MDPs. We first presented three algorithms for this based on strategy improvement, fractional linear programming and linear programming. We then compared performance characteristics of these algorithms and integrated the best performing one into the probabilistic model checker PRISM. Building upon this, we introduced a semi-symbolic policy iteration algorithm and reported on experiments with its integration into PRISM.

Future Work. There still remains work to do. Of interest are methods to scale the existing algorithms to even larger MDPs. For example, parallelization of the algo-

rithms could be considered. In others direction, a variety of techniques for abstraction and decomposition of models to obtain smaller models have been developed, which might prove beneficial in our setting. Finally, a considerable bottleneck in our implementation is the decomposition into end components: faster algorithms in this area (such as [10]) would improve performance significantly.

References

1. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.
2. C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 430–440. Springer, 1997.
3. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
4. R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 140–156. Springer, 2009.
5. R. Bloem, H. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Proc. 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, pages 37–54, 2000.
6. R. Bloem, K. Greimel, T. A. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *FMCAD*, pages 85–92. IEEE, 2009.
7. T. Brázdil, V. Brožek, K. Chatterjee, V. Forejt, and A. Kučera. Two views on multiple mean-payoff objectives in Markov decision processes. In *LICS*, pages 33–42. IEEE Computer Society, 2011.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
9. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
10. K. Chatterjee and M. Henzinger. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'11)*, pages 1318–1336, 2011.
11. K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 380–395. Springer, 2010.
12. K. Chatterjee, R. Majumdar, and T. Henzinger. Markov decision processes with multiple objectives. In *Proc. 23rd International Symposium on Theoretical Aspects of Computer Science (STACS'06)*, volume 3884 of *LNCS*, pages 325–336. Springer, 2006.
13. L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
14. C. Derman. On sequential decisions and Markov chains. *Management Science*, 9(1):16–24, 1962.
15. K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 50–65. Springer, 2007.
16. V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 112–127. Springer, 2011.
17. M. Fujita, P. C. Mcgeer, and J. C. Y. Yang. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, V10(2/3):149–169, April 1997.
18. H. Gimbert. Pure stationary optimal strategies in Markov decision processes. In *STACS'07*, pages 200–211. Springer-Verlag, 2007.

19. G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(12):1479 – 1493, dec 1996.
20. B. R. Haverkort. *Performance of computer communication systems - a model-based approach*. Wiley, 1998.
21. J. R. Isbell and W. H. Marlow. Attrition games. *Naval Research Logistics Quarterly*, 3:71–94, 1956.
22. J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. Springer-Verlag, 2nd edition, 1976.
23. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.
24. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
25. M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.
26. M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.
27. D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, 1981.
28. Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
29. G. Norman, D. Parker, M. Kwiatkowska, S. K. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Asp. Comput.*, 17(2):160–176, 2005.
30. J. Norris. *Markov Chains*. Cambridge University Press, 2003.
31. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
32. R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *NIPS*. The MIT Press, 1997.
33. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
34. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.
35. M. O. Rabin. N-process mutual exclusion with bounded waiting by 4 $log_2 n$-valued shared variable. *J. Comput. Syst. Sci.*, 25(1):66–75, 1982.
36. H. C. Tijms. *A First Course in Stochastic Models*. Chichester: Wiley, 2003.
37. C. von Essen and B. Jobstmann. Synthesizing systems with optimal average-case behavior for ratio objectives. In J. Reich and B. Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 17–32, 2011.
38. C. von Essen and B. Jobstmann. Synthesizing efficient controllers. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 428–444, 2012.
39. R. Wimmer, B. Braitling, B. Becker, E. M. Hahn, P. Crouzen, H. Hermanns, A. Dhama, and O. Theel. Symblicit calculation of long-run averages for concurrent probabilistic systems. In *QEST*, pages 27–36. IEEE Computer Society, 2010.
40. R. Wimmer, S. Derisavi, and H. Hermanns. Symbolic partition refinement with dynamic balancing of time and space. In *QEST*, pages 65–74. IEEE Computer Society, 2008.
41. H. Yue, H. C. Bohnenkamp, and J.-P. Katoen. Analyzing energy consumption in a gossiping MAC protocol. In B. Müller-Clostermann, K. Echtle, and E. P. Rathgeb, editors, *MMB/DFT*, volume 5987 of *LNCS*, pages 107–119. Springer, 2010.